TFE: DESARROLLO DISPOSITIVO DE COMUNICACIÓN CON KINECT 2.0

GRADO INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

AUTOR: ROMÁN SÁNCHEZ SIERRA

TUTOR: FERNANDO GARCÍA FERNÁNDEZ

TÍTULO: Desarrollo de dispositivo de comunicación con Kinect 2.0

AUTOR: Román Sánchez Sierra
TUTOR: Fernando García Fernández

EL TRIBUNAL

Presidente:

Vocal:

Secretario:

Realizado el acto de defensa y lectura del Trabajo Fin De Grado el día 6 de octubre de 2016 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de:

PRESIDENTE

VOCAL

SECRETARIO

Agradecimientos

Después de cuatro años de duro trabajo me dispongo a dar por concluida una etapa más de mi vida. Con la realización de este Trabajo Fin de Grado puedo dar por finalizada mi etapa universitaria pero no formativa, pues un ingeniero nunca deja de crecer profesionalmente.

En primer lugar quiero dar las gracias a mi familia por el apoyo recibido. En especial a mis padres que día tras día me han tendido la mano y no han dejado que me rinda a pesar de los baches superados en el camino.

En segundo lugar agradecer a mi novia y a mis amigos la paciencia que han tenido y los ánimos brindados en los peores momentos, en el desarrollo de este proyecto y por haberme acompañado en los momentos más importantes.

Agradecer también a los compañeros del 'Intelligent Systems Laboratory' por ayudarme a sacar adelante el proyecto.

Por último, pero no menos importante, quiero darle las gracias a mi tutor Fernando García Fernández haberme dado la oportunidad de realizar este proyecto. Además, quiero agradecerle la formación y la ayuda brindada tanto en el proyecto como en las asignaturas en las cuales ha sido mi profesor.

A todos ellos, gracias.

Resumen

Debido al desarrollo y crecimiento de los coches autónomos en la actualidad, surge la necesidad de desarrollar un sistema de detección en tiempo real que permita al vehículo detectar obstáculos y/o personas así como permitir la posibilidad de planificar rutas que permitan una navegación segura.

A día de hoy existen múltiples algoritmos de detección de obstáculos y generación de trayectorias desarrollados por las empresas mas potentes como Google, Tesla o BMW. Se trata de un campo de gran crecimiento en la industria del automóvil.

Sin embargo, el propósito de este proyecto será crear un algoritmo sencillo y con un presupuesto asequible que permita realizar las funciones anteriormente descritas.

Por tanto, el presente proyecto se basa en el desarrollo de un sistema que genera un mapa bidimensional en el cual se representan los obstáculos detectados por medio de las imágenes obtenidas por una cámara situada en el vehículo.

Para alcanzar el objetivo descrito, se usa una Kinect 2.0 desarrollada por Microsoft para Xbox One. Ésta será conectada a una placa de desarrollo Odroid XU4. El sistema se implementará en ROS Jade y la detección de obstáculos se realiza analizando nubes de puntos gracias a las librerías PCL.

El resultado del proyecto será un tipo de escaneo laser obtenido por medio de las imágenes del entorno capturadas por la Kinect. Se trata por tanto de un mapa donde se pueden localizar los obstáculos encontrado así como el espacio libre por donde pueda navegar el vehículo de forma autónoma.

Abstract

Due to development and growth of autonomous cars today, it is necessary to develop a real-time detection system which allows the vehicle to detect obstacles or people. The system should allow to plan routes and navigate safely.

Nowadays there are several algorithms able to detect obstacles and to plan routes. These algorithms are developed by some of the most powerful companies such as Google, Tesla or BMW. It is a field of great growth in the automotive industry.

However, the purpose of this project is to develop a simple algorithm with low budget which should allow all described functions. Therefore, this project is based which on the development a two-dimensional map where detected obstacles will be represented.

To obtain images of the environment Kinect 2.0 is used, developed by Microsoft. This camera will be connected to a development board Odroid XU4. Detection system is implemented in ROS Jade. Obstacles detection is schedule with PCL libraries, so it is necessary to analyse point clouds.

The result is a type of laser scan, obtained through pictures of environment. This scan is a map where obstacles are represented. Also, free space where vehicle can navigate automatically is represented.

ÍNDICE

CAPÍTULO 1: INTRODUCCIÓN 11

1. INTRODUCCIÓN 12

1.1. DESCRIPCIÓN DEL PROYECTO 12

CAPÍTULO 2: COCHES AUTÓNOMOS EN LA ACTUALIDAD 14

1. ESTADO DEL ARTE DE LOS COCHES AUTÓNOMOS 15

1.1. EL COCHE AUTÓNOMO DE GOOGLE 15

1.2. EL COCHE AUTÓNOMO COMO OPORTUNIDAD DE NEGOCIO 17

1.3. LEGISLACIÓN SOBRE COCHES AUTÓNOMOS 17

1.4. VISIÓN ESTÉREO BASADA EN MAPAS LOCALES DE OCUPACIÓN PARA NAVEGACIÓN AUTÓNOMA EN ROS 18

1.4.1. RESUMEN 18

1.4.2. INTRODUCCIÓN 19

1.4.3. DESCRIPCIÓN DE LOS SISTEMAS 19

1.4.4. ALGORITMOS 20

1.4.5. IMPORTANCIA PARA EL PROYECTO 21

CAPÍTULO 3: 23

DESCRIPCIÓN DE HARDWARE Y SOFTWARE 23

1. DESCRIPCIÓN DE HARDWARE Y SOFTWARE 24

1.1. PLACA DE DESARROLLO ODROID XU4 24

1.2. KINECT 2.0. 25

1.3. ROS – ROBOT OPERATING SYSTEM 26

1.3.1. Introducción a ROS 26

1.3.2. ¿Por qué ROS? 27

1.4. DRIVERS KINECT 2.0 27

1.5. IAI KINECT2 27

1.6. DEPENDENCIAS PARA TRABAJAR CON SENSOR KINECT 28

1.7. POINT CLOUD LIBRERY (PCL) 28

CAPÍTULO 4: FASES DEL PROYECTO 29

1. FASES DEL PROYECTO. 30

1.1. FASE 1: APRENDIZAJE 30

1.2. FASE 2: CONFIGURACIÓN. 31
1.3. FASE 3: DESARROLLO 31

CAPÍTULO 5: APRENDIZAJE Y CONFIGURACIÓN 34

1. FASE 1: AUTOAPRENDIZAJE. 35
1.1. ROS 35
1.1.1. Sistema de ficheros ROS. 35
1.1.2. Elementos de computación 36
1.1.3. Concepto de nodo 36
1.1.4. Concepto de topic 37
1.2. LIBRERÍAS PCL 37

2. FASE 2: CONFIGURACIÓN. 39
2.1. INSTALAR IMAGEN UBUNTU EN SD O EMMC. 39
2.2. CONFIGURACIÓN ODROID. 40

CAPÍTULO 6: DESARROLLO 41

1. FASE 3: DESARROLLO 42
1.1. CREAR EL PAQUETE DEL PROGRAMA. 43
1.2. DESCRIPCIÓN DEL PROGRAMA. 45
1.2.1. Librerías necesarias 45
1.2.2. Función main 48
1.2.3. Función Callback del nodo 50
1.2.4. Detección y eliminación del plano suelo 51
1.2.5. Calibración Kinect 52
1.2.6. Filtrar la nube en función del intervalo calculado 53
1.2.7. Transformación de la nube 54
1.2.8. Publicar la nube 55
1.2.9. Generar escaneo laser: función PCL_to_laserscan 55

CAPÍTULO 7: PRUEBAS REALIZADAS 62

1. PRUEBAS REALIZADAS 63
1.1. DETECCIÓN DEL SUELO 63
1.2. ELIMINACIÓN DE LOS PUNTOS DEL SUELO 64
1.3. PRUEBA COMPLETA EN EL LABORATORIO 65
1.4. PRUEBA COMPLETA EN EL LABORATORIO II 67
1.5. PRUEBA EN EXTERIORES 69

1.5.1. Configuración RVIZ para visualización 69

1.5.2. Ejecución y visualización 71

CAPÍTULO 8: CONCLUSIONES Y PRESUPUESTO 76

1.1. CONCLUSIONES 77

1.2. APLICACIONES 77

1.3. PRESUPUESTO 78

ANEXOS 79

ANEXO I: ENTENDER NODOS Y TOPICS: EJEMPLO TURTLESIM 80

ANEXO II: METODO RANSAC 81

1. DEFINICIÓN 81

2. VENTAJAS Y DESVENTAJAS DE RANSAC 81

BIBLIOGRAFÍA 83

ÍNDICE DE ILUSTRACIONES

Ilustración 1: Placa de desarrollo Odroid XU4 24
Ilustración 2: Kinect 2.0 25
Ilustración 3: Instalación de Ubuntu 14.04 en SD con Apple Pi Baker 39
Ilustración 4: Situación de tarjeta eMMC en Odroid 40
Ilustración 5: Diagrama del software 42
Ilustración 6: Contenido del paquete Kienct_Cloud en el workspace de ROS 43
Ilustración 7: Contenido del paquete Kinect_Cloud 44
Ilustración 8: Ubicación del archivo .cpp 44
Ilustración 9: Calculo intervalo de puntos que se van a utilizar 53
Ilustración 10: Resultado prueba 1 detección de suelo 63
Ilustración 11: Resultado prueba 2 detección de suelo 64
Ilustración 12: Resultado prueba eliminación del suelo 64
Ilustración 13: Disposición de Kinect para prueba en laboratorio 65
Ilustración 14: Imagen de profundidad durante la prueba 66
Ilustración 15: Nube de puntos obtenida durante la prueba 66
Ilustración 16: Mapa obtenido durante la prueba 67
Ilustración 17: Prueba de detección de personas 68
Ilustración 18: Configuración Display de tipo PointCloud 70
Ilustración 19: Configuración Display de tipo LaserScan 70
Ilustración 20: Plano Campus de Leganés 71
Ilustración 21: Zona de pruebas 72
Ilustración 22: Detección de pared 70
Ilustración 23: Detección de esquina 71
Ilustración 24: Detección de espacio libre 71
Ilustración 25: Detección de persona I 72
Ilustración 26: Detección de persona II 72

CAPÍTULO 1:

INTRODUCCIÓN

1. INTRODUCCIÓN

El objetivo de este proyecto es desarrollar un sistema de captura y análisis de imagen usando el sensor Kinect 2.0, desarrollado para Xbox. Este sistema de comunicación será utilizado en un coche autónomo, lo que permitirá reconocer el entorno en el que se encuentra.

El problema del reconocimiento ha sido tratado en muchas ocasiones. Inicialmente se analizaban imágenes en 2D que eran proporcionadas por una cámara digital. Estas imágenes impedían identificar la profundidad de los objetos vistos en ellas. En un primer momento se intentó solucionar el problema utilizando varias cámaras localizadas en diferentes posiciones. Esto permitía obtener imágenes en 3D.

Con el lanzamiento de Kinect cambió el proceso de reconocimiento de imágenes. Kinect es un sensor barato y muy efectivo ya que obtiene imágenes en 3D.

1.1. Descripción del proyecto

Este proyecto esta dedicado a desarrollar un dispositivo de comunicación con Kinect 2.0. El objetivo del mismo, es obtener un mapa 2D del entorno en el cual se sitúa la Kinect. El fin de este proyecto es su utilización en un coche autónomo. De ello viene la necesidad de generar un mapa 2D, con el que se permitirá al coche saber donde se encuentra y reconocer los posibles obstáculos.

A nivel de hardware, se ha elegido un sistema embebido para desarrollar el proyecto. Concretamente la placa de desarrollo XU4, muy similar a Raspberry Pi pero con unas características más sofisticadas. A la placa irá conectada la Kinect mediante USB. Es necesario un puerto USB 3.0. Esta es la razón por la que se ha elegido la placa de desarrollo utilizada.

En cuanto al software, se trabaja con Ubuntu 14.04. Es necesario instalar ROS, un sistema operativo desarrollado específicamente para robots. La versión elegida en este proyecto en Jade.

Además de Ubuntu y ROS, es necesario instalar los driver de Kinect, libfreenect2. Permitirán usar el sensor en el sistema Ubuntu. Una vez que Kinect funciona correctamente en Ubuntu es necesario permitir su uso en ROS. Para ello

se usará IAI_Kinect2. Se trata de un conjunto de librerías y herramientas que actúan a modo de puente entre Ubuntu y Ros.

Una vez hecho esto, será el momento de desarrollar el programa. Se incluye dentro de un paquete ROS llamado *Kinect_cloud*. Este programa deberá obtener una nube de puntos, proporcionada por Kinect, procesarla y publicarla. El resultado que se publica en ROS será una nube previamente filtrada en función de un intervalo que dependerá de la altura a la cual se coloque la Kinect.

Cuando la nube haya sido publicada en ROS, será usada por una tercera función llamada *PCL_to_LaserScan.* Esta función transformará la nube de puntos simulando un barrido laser. El resultado de esta operación será un mapa 2D similar al que se obtendría con un laser.

CAPÍTULO 2:

COCHES AUTÓNOMOS EN LA ACTUALIDAD

1. ESTADO DEL ARTE DE LOS COCHES AUTÓNOMOS

En la actualidad, se habla con bastante frecuencia de coches que se mueven sin necesidad de conductor. Se trata de una idea que, no hace mucho, parecía ciencia ficción y que a día de hoy esta muy cerca de convertirse en realidad. Pero, ¿qué es un coche autónomo y cómo funciona?

Por definición, un vehículo autónomo es un automóvil capaz de imitar las capacidades humanas de manejo y control, percibiendo el medio que le rodea y desplazándose en consecuencia. Es decir, un coche en el que el 'conductor' solo tiene que introducir la dirección de destino y despreocuparse de todo lo demás.

El origen de los coches autónomos es mucho más remoto de lo que se suele pensar. Desde los años 40 se han estado haciendo pruebas de guiado de vehículos embebiendo dentro del asfalto materiales que podían ser detectados y seguidos. De ahí se ha pasado a la detección de obstáculos vía radar en los años 80 y en la actualidad la tecnología es un complejo conjunto de tecnologías que incluyen el reconocimiento de movimiento por cámaras y los sistemas de detección láser.

Actualmente, se pueden ver circulando coches automáticos; con sistemas de ayuda al aparcamiento – incluso que aparcan solos; ayudas a la conducción, tales como asistente de colisión, ESP, control de par, asistente de cambio de carril, detección de ángulos muertos… Los coches actuales incorporan incorporan un gran equipamiento tecnológico pero la ingeniería y el sector automovilístico quieren dar un paso más.

1.1. El coche autónomo de Google

Google ha sido una de las primeras empresas en desarrollar un coche autónomo. Poniéndolo como ejemplo se puede entender como es el funcionamiento de estos vehículos.

El coche autónomo de Google conduce solo, reconoce carriles, señales de tráfico, semáforos, cruces… Tiene capacidad para reconocer otros vehículos, ciclistas, peatones… Controla le distancia con el vehículo de delante y toma las decisiones pertinentes para no ocasionar un accidente. En definitiva, incorpora todas las capacidades de un conductor humano. Pero, ¿qué dispositivos utiliza para ello?

El vehículo incorpora un lidar con el cual genera una imagen tridimensional en la cual se ven los obstáculos que hay en el entorno. Este sistema se complementa con un GPS y una unidad de medición inercial. Con ello se puede determinar hacia donde se mueve el coche.

Además, dispone de cuatro radares: tres colocados en el paragolpes delanteros y otro en el trasero. Los tres radares delanteros calculan la distancia con el vehículo de delante y detecta el carril o coches estacionados. El trasero actúa como espejo retrovisor y se usa cuando se circula marcha atrás.

En la parte alta del parabrisas, centrada, hay una cámara que reconoce las señales de tráfico, los semáforos y las líneas de la calzada. Y finalmente el último sensor es un codificador en la rueda trasera izquierda, que mide con precisión la distancia recorrida, determina la ubicación exacta del coche y sigue los movimientos del mismo.

Hay un último elemento que también ayuda a que el coche autónomo de Google funcione tan bien por sí solo: al menos durante el período de pruebas, todos los recorridos que hace el coche los han realizado al menos una vez antes los ingenieros.

Toda la información relativa al recorrido queda registrada y cuando el coche vuelve a realizar ese recorrido por sí mismo, compara los datos que recogen los sensores con los grabados. Así se reconoce lo que es un árbol, una farola o un buzón, de lo que es un peatón.

El acelerador, el freno y la dirección ya se pueden controlar de manera automática con electrónica, en el fondo no es algo tan nuevo, aunque todavía no se haya generalizado. Un sistema de control de velocidad de crucero y acelerador electrónico permite que el computador acelere el motor más o menos según su criterio.

El sistema de frenos también se puede controlar con un accionador eléctrico, e igualmente la dirección, donde un motor eléctrico hace girar el volante tantos grados en uno u otro sentido. En los sistemas de precolisión y frenado automático, o en los sistemas de asistente de estacionamiento ya se implementan (aunque suelen ser equipamientos opcionales).

1.2. El coche autónomo como oportunidad de negocio

El coche autónomo generará nuevas oportunidades de negocio, así como el desarrollo de nuevos servicios de asistencia al conductor, según la aseguradora Mapfre. Desde la compañía han manifestado que las oportunidades del coche autónomo vendrán en el ámbito de la ciberseguridad, que será imprescindible a la hora de prevenir o evitar ataques remotos por parte de hackers al sistema operativo de los coches, desde los que ya se puede controlar el navegador, acelerador o los frenos. El coche autónomo generará nuevas oportunidades de negocio y permitirá al sector asegurador prestar nuevos servicios al cliente con el fin de ofrecerles asistencia temprana en caso de accidente y localización del vehículo en caso de robo.

El coche autónomo generará nuevas oportunidades de negocio pero también desarrollará nuevos sistemas de asistencia al conductor como el control de velocidad, el sistema de ayuda a la frenada y mantenimiento de carril, basados en sensores de entorno, como radares, vídeo o ultrasonidos, y que según algunos estudios supondrá una reducción significativa de la siniestralidad a medida que vaya aumentando su implantación en el parque móvil.

En cuanto a la legislación de los vehículos autónomos, Jaime Moreno García-Cano, subdirector General de la Dirección General de Tráfico, ha analizado la visión jurídica o legislativa de los vehículos autónomos. En este sentido ha indicado que actualmente la circulación de vehículos autónomos está regulada dentro del marco de las pruebas de ensayos e investigación que recoge el Reglamento General de Vehículos y que en base a esta normativa, la DGT aprobó el año pasado una instrucción específica donde se recoge el proceso para la obtención de la autorización de pruebas para la conducción autónoma en España en vías abiertas al tráfico en general.

1.3. Legislación sobre coches autónomos

El coche autónomo es una de los temas del momento. Compañías como Google, Ford, Tesla e incluso Samsung han desarrollado sus prototipos para este tipo de vehículos, prototipos que ya están empezando probarse.

A medida que el desarrollo del coche autónomo ha ido avanzando, el problema de la regulación y normas para su uso aun está por resolver. El Gobierno de California ha sido el primero en lanzar un primera propuesta para ello, advirtiendo de que pueden sucederse numerosos cambios en la legislación. Esta ley entró en vigor el pasado 1 de enero de 2016.

El Gobierno ha especificado que todos los coches autónomos deberán tener pedales, volante y un conductor con licencia. Estas tres normas ya habían sido notificadas a Google cuando probó su prototipo en el estado de California.

Sin duda el punto mas importante que se ha incluido en la legislación ha sido que los coches autónomos no podrán ser vendidos. Las compañías deberán ofrecer sus vehículos en arrendamiento durante tres años. Además, las compañías deberán obtener una licencia que deberá ser solicitada al Departamento de Vehículos Automotores (DMV).

El cuanto al usuario que desee obtener un coche autónomo, deberá obtener una licencia expedida por el mismo departamento, además de realizar un entrenamiento especial por parte de la compañía que oferta el vehículo.

1.4. Visión estéreo basada en mapas locales de ocupación para navegación autónoma en ROS

1.4.1. RESUMEN

La navegación autónoma para vehículos terrestres no tripulados ha obtenido un significativo interés en la comunidad de investigación de robots móviles. Esta mayor atención se debe a que tiene un papel considerable dentro del campo de los Sistemas de Transporte Inteligentes – del inglés ITS. Con el fin de conseguir la navegación autónoma para vehículos terrestres, son requeridos modelos del entorno detallados, que se tratarán como mapas. En este apartado se presenta una innovadora aproximación en reconocer obstáculos inmóviles, por medio de cámaras estéreo, y construir un mapa local de ocupación en ROS. Los mapas incluyen información con respecto a estructuras 3D del entorno, que se obtienen con cámaras estéreo. Estos mapas pueden mejorarse con mayor detalle de los obstáculos no detectados por un telémetro laser. A fin de evaluar la propuesta aproximada, se han realizado varios ensayos en diferentes escenarios. Los mapas

obtenidos son comparados exactamente con la parte correspondiente del mapa global y con la imagen satélite equivalente. Los resultados obtenidos indican el alto rendimiento en numerosas situaciones.

1.4.2. INTRODUCCIÓN

Durante la última década el número de robots móviles en el mercado ha tenido un rápido crecimiento, relacionado con complicadas tareas como navegación autónoma. Estas tareas hacen necesario un amplio conocimiento de los alrededores. Hoy en día, existen varios métodos para obtener información de entorno. Desde un punto de vista del sensor, se pueden dividir en dos grupos. Primero, hay suficientes algoritmos basados en laser, que aportan medidas de alta precisión aunque a veces no aportan suficiente información para clasificar los elementos del entorno. Además, los sensores basados en visión artificial son sistemas ricos en información, especialmente la visión estéreo, acosta de una peor precisión. Los algoritmos de visión estéreo son claves para detectar obstáculos y espacio libre delante del vehículo; concretamente, muchos autores representas esta información como el mapa de obstáculos y el mapa libre. Respecto a la navegación autónoma, los obstáculos y el espacio libre serán representados en un mapa de ocupación.

Es esencial estimar la posición de la cámara respecto al entorno con el fin de obtener unas medidas lo más reales posibles. Por ejemplo, un patrón de calibración puede colocarse en el suelo en frente del vehículo fuera de la línea de estimación. Sin embargo este método de calibración no permite actualizar posiciones de la cámara mientras el vehículo se mueve. Esto se resuelve colocando puntos de referencia en el entorno.

La principal contribución para este trabajo propone un robusto e instantáneo mapa de ocupación.

1.4.3. DESCRIPCIÓN DE LOS SISTEMAS

Para este trabajo, se han realizado pruebas en un vehículo autónomo terrestre llamado iCab (**I**ntelligent **C**ampus **A**utomo**b**ile). Se trata de un carrito de golf

modificado mecánica y electrónicamente para satisfacer los requisitos de la navegación autónoma.

El vehículo tiene actuadores electrónicos que permiten su movimiento. El pedal del acelerador ha sido sustituido por un circuito amplificador de potencia que controla la tracción trasera y delantera. Además, el vehículo esta equipado con múltiples sensores que permiten obtener información del entorno.

Para el control del vehículo se encuentra instalado un ordenador Intel, que corre sobre Ubuntu. Todos los algoritmos necesarios están implementados en ROS (Robot Operating System), con el fin de comunicar los diferentes procesos y fusionar la información de los múltiples sensores.

1.4.4. ALGORITMOS

Para generar el mapa de ocupación se usa la información de la cámara estéreo. Esta información se corresponde, en primer lugar, con la estimación de la posición de la cámara y con los obstáculos y espacio libre delante del vehículo.

La posición de la cámara se estima en función de los parámetros extrínsecos respecto al entorno. En otras palabras, se estima por los valores de la altura, ángulo de cabeceo y el ángulo de rotación. Estos valores son necesarios para realizar la calibración de la cámara.

La detección de obstáculos y espacio libre se realiza por medio del mapa de disparidad. Se obtiene a a través de las imágenes estéreo y contiene información sobre profundidad, donde el valor de cada pixel corresponde con la disparidad.

Mapa de ocupación

Las cuadrículas del mapa se genera, tan pronto como se obtengan los obstáculos y el espacio libre, junto con los parámetros extrínsecos de la cámara. A continuación, se divide en diferentes regiones de interés llamadas Regions of Interest (ROIs).

Las celdas del mapa se obtienen de cada ROI, indicando que obstáculos son mas fáciles de manejar. Se asigna un valor único de disparidad a cada región. Este valor corresponde con el modo de disparidad para cada pixel de un obstáculo.

Un obstáculo es segmentado y asociado a un valor de disparidad. Las coordenadas reales se obtienen de las siguientes ecuaciones:

$$X = \frac{b \cdot \cos\theta \cdot \sin\rho \cdot (\vartheta - \vartheta_0) + b \cdot \cos\rho(u - u_0) + \alpha b \cdot \sin\rho \cdot \sin\theta}{\Delta}$$

$$Y = -h + \frac{b \cos\rho \cos\theta(\vartheta - \vartheta_0) - \sin\rho(u - u_0) + \alpha b \cos\rho \sin\theta}{\Delta}$$

$$Z = \frac{\alpha b \cos\theta - b \sin\theta(\vartheta - \vartheta_0)}{\Delta}$$

Para cada celda del mapa, la coordenada Y se calcula a partir de su esquina inferior izquierda y clasificada entre dos categorías: en el suelo o elevada. Si la elevación del obstáculo es mayor que un límite especificado, cercano a cero, se considera que está elevado. Este algoritmo asume que la distancia de todos los pixeles del obstáculo a la cámara es la misma. Por lo tanto, los obstáculos cercanos al sensor se deben fragmentar en diferentes regiones, según sus niveles de disparidad.

Una vez creado el mapa de celdas se publica en un mensaje de ROS de tipo OcucpancyGrid. Este tipo de dato se considera estándar según la comunidad ROS. Cada pixel del mapa representa un cierto área del mundo real con respecto a la resolución de las celdas.

1.4.5. IMPORTANCIA PARA EL PROYECTO

Como se puede ver, existe una clara similitud entre un mapa de ocupación y el escaneo laser que generará este proyecto. Para ambos son necesarios los parámetros extrínsecos de la cámara así como una calibración del sensor con el cual se obtiene la información del entorno.

En cuanto a la información de los obstáculos, en ambos casos se considera que todos los pixel del obstáculo están a la misma distancia de la cámara. Las coordenadas reales de los mismo se calculan de diferente manera en cada método. En el caso de mapas de ocupación se obtienen por las formulas vistan anteriormente. En el escaneo laser se obtienen en función de la posición respecto a la cámara y la orientación de la misma, haciendo uso de la trigonometría.

A la hora de publicar la información obtenida, se realiza de forma diferente, aunque ambas se publiquen bajo un mensaje de ROS. La diferencia es que en

mapas de ocupación se publica un tipo de mensaje OccupancyGrid y en escaneo laser, el tipo de mensaje publicado es LaseScan.

CAPÍTULO 3:

DESCRIPCIÓN DE HARDWARE Y SOFTWARE

1. DESCRIPCIÓN DE HARDWARE Y SOFTWARE

Como ya se ha introducido, el proyecto persigue el objetivo de obtener y analizar imágenes obtenidas con el sensor Kinect 2.0. Para ello es necesario utilizar ROS, ya que nos permite comunicarnos con la Kinect y conseguir las imágenes que posteriormente analizaremos. Se puede decir que se pretende desarrollar un sistema de visión artificial que podría usarse en aplicaciones robóticas o incluso en sistemas de seguridad. ROS correrá sobre la versión 14.04 de Ubuntu/Linux, que será instalada en una placa de desarrollo: Odroid XU4.

Para poder usar Kinect será necesario instalar también sus drivers – libfreenect2. Además, este dispositivo debe ser conectado a un USB 3.0 para que funcione correctamente. Esto no es problema, pues la placa de desarrollo Odroid XU4 incorpora dicho puerto USB.

1.1. Placa de desarrollo Odroid XU4

Se trata de una placa de desarrollo similar a la Raspberry Pi. El Odroid XU4 es mucho más potente y cuenta con más opciones que la Rapsberry.

Odroid XU4 incorpora un procesador Samsung Exynos 5422 de 8 núcleos cuyo funcionamiento tiene una velocidad máxima de 2GHz. Monta además una memoria RAM LPDDR3 de 2Gb y un controlador de memoria flash eMMC de 8 bits.

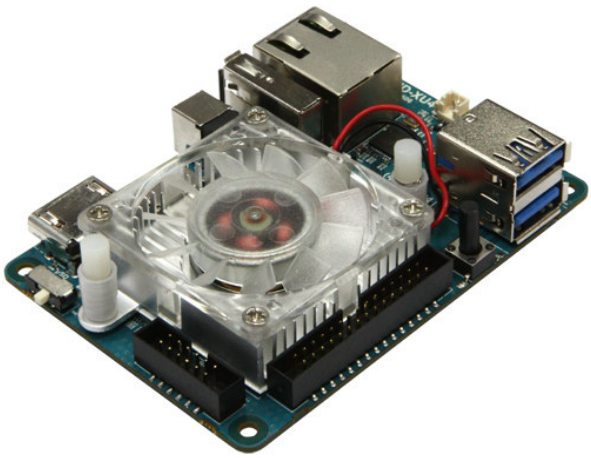


Ilustración 1: Placa de desarrollo Odroid XU4

Se ha elegido este dispositivo debido a que incorpora dos puerto USB 3.0, necesarios para conectar la Kinect y que funcione correctamente. Además tiene una salida HDMI que nos facilita mucho su uso si conectamos Odroid XU4 a un monitor.

En el proceso de configuración e instalación de todo el software necesario para llevar a cabo el proyecto, se ha encontrado una imagen de Ubuntu 15.04 que ya incorpora los drivers de Kinect2 y ROS Jade instalado, además de las librerías OpenCV y PCL. Esta imagen se puede descargar desde el siguiente enlace:

http://forum.odroid.com/viewtopic.php?f=95&t=16149

1.2. Kinect 2.0.

Kinect es una cámara desarrollada por Microsoft para darla uso en Xbox One. Esta segunda versión, la que se usará en este proyecto, incorpora la nueva cámara TOF (Time-Of-Flight) que ofrece una resolución de 1080p y mucho más detalle a la hora de capturar movimiento. Cuenta con un campo de visión un 60% más grande que la versión uno.

Kinect2 incorpora un nuevo sensor infrarrojo que permite detectar personas y objetos en una habitación totalmente a oscuras. Sumado a la nueva cámara permite obtener información sobre detalles del rostro, esqueleto, los músculos y el latido del corazón de las personas que están frente al sensor.

Ilustración 2: Kinect 2.0

Esta versión es mucho más potente que su antecesora. Son muchas las diferencias que existen entre Kinect 2.0 y la versión anterior. Esto abre un gran abanico de posibilidades de desarrollo.

- **Mayor campo de visión.** 70º en horizontal y 60 en vertical. En la versión anterior eran 57 y 43 respectivamente. Esto permite detectar a mas personas dentro de un mismo campo de visión.
- **Mayor resolución, 1920x1080 Full HD** mientras que antes era de 640x480. Permite detectar con más precisión todo el entorno. Tiene mayor capacidad para diferenciar la orientación del cuerpo incluyendo manos e incluso dedos. A esto hay que sumar el reconocimiento de gestos de caras.
- **Mejora el rango de profundidad del sensor.** El rango de actuación pasa de ser 0,5 a 4,5 metros.
- **USB 3.0.** Al aumentar la velocidad de comunicación los datos fluyen más rápido, disminuyendo así la latencia del sensor. Pasa de 90ms a 60ms .
- **Mejora la captación de sonidos.** Esta versión de Kinect tiene una gran capacidad de reconocimiento y captación de sonidos. Se elimina el ruido ambiente, lo que permite captar con mas detalle las instrucciones vocales. Este punto no es de mucha importancia en el desarrollo del proyecto, pero es una mejora más que incorpora Kinect 2.0.
- **Captación de movimientos a oscuras.** Ahora Kinect es capaz de reconocer y captar movimientos incluso a oscuras.

1.3. ROS – Robot Operating System

1.3.1. Introducción a ROS

Se trata de un framework para el desarrollo de software y aplicaciones de robots. Fue desarrollado en 2007 por el Laboratorio de Inteligencia Artificial de Standford con el fin de dar soporte al robot conocido como Stair. Hay otros robots conocidos que también trabajan con ROS, por ejemplo PR1, PR2 o Robot de Shadow. En este último participo la Universidad Carlos III de Madrid.

ROS proporciona los servicios estándar de un sistema operativo como atracción de hardware, control de dispositivos a bajo nivel, funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes.

Su funcionamiento se basa en nodos que pueden recibir y enviar mensajes referentes a sensores, control, estados, planificadores y actuadores, entre otros. Aunque está orientada a Linux, ROS puede correr en Windows o Mac OS X.

Tiene dos partes básicas: la parte del sistema operativo, ros y ros-pkg, y una colección de paquetes aportados por la comunidad de usuarios (organizados en conjuntos llamados stacks) que implementan funcionalidades tales como SLAM, planificación, percepción, simulación, etc.

ROS está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. En este proyecto permitirá comunicarnos con la Kinect y obtener imágenes a través del nodo Kinect_node.

1.3.2. ¿Por qué ROS?

Un motivo por el cual se ha elegido ROS es su gran modularidad ya que permite usar las partes que se desee e implementar las que se necesite. La gran comunidad de la que dispone ROS ha sido el segundo motivo por el cual se ha elegido. Es fácil encontrar software dedicado a hacer las tareas que se quieren realizar.

1.4. Drivers Kinect 2.0

Para poder usar la kinect y visualizar el entorno con ella es necesario tener sus drivers instalados. Además, usaremos un medio por el cual podamos obtener los datos que publica la kinect.

Libfreenect2 son los drivers de kinect para Linux. No será necesario instalarlo pues ya viene incluido en la imagen que usaremos para Odroid. Estos drivers nos darán la información que capture la kinect. Para tener acceso a ellos usaremos IAI Kinect2. Este software hará de puente entre ROS y Kinect.

1.5. IAI Kinect2

Se trata de una colección de herramientas y librerías para ROS y Kinect. Contiene:

- Herramienta de calibración para calibrar el sensor infrarrojo – IR sensor – de Kinect al sensor RGB y las correspondientes mediciones de profundidad.
- Una librería para el registro de profundidad con soporte para OpenCL.
- Una conexión entre libfreenect2 y ROS.
- Visor de imágenes y point clouds – nubes de puntos.

En cuanto la instalación, hay que tener previamente instalado y configurado ROS, asi como el driver libfreenect2. Una vez hecho esto, se puede seguir el tutorial del siguiente repositorio a partir del paso 5.

https://github.com/code-iai/iai_kinect2

1.6. Dependencias para trabajar con sensor Kinect

Existen una serie de dependencias que permiten usar la Kinect con ROS en la placa de desarrollo Odroid. Su instalación es algo costosa ya que se han tenido que instalar y compilar una a una estas dependencias.

Como no se sabe cuando puede suceder un problema y perder la configuración realizada hasta el momento, se ha optado por hacer una imagen del sistema operativo. De esta manera, ante cualquier imprevisto, solo será necesario reinstalar el sistema desde esta imagen y no haría falta volver a configurar de nuevo.

Tanto el paquete de descarga de las dependencias como la explicación de cómo hacer una imagen de un sistema operativo se anexan al final de la memoria.

1.7. Point Cloud Librery (PCL)

Point Cloud Librery es un proyecto abierto independiente y a gran escala para el procesamiento punto a punto de nubes en 2D y 3D. PCL está publicada bajo los términos de licencia BSD y por tanto se trata de una librería libre para uso comercial y de investigación.

En el siguiente enlace se puede acceder a la documentación de esta librería donde se pueden realizar los tutoriales e ir conociendo esta herramienta.

http://pointclouds.org/documentation/

Estas librerías permitirán conseguir una nube de puntos que defina el entorno en el que se encuentre el coche autónomo. El objetivo será tratar y filtrar dicha nube para tener unos datos correctos acerca del entorno.

CAPÍTULO 4:

FASES DEL PROYECTO

1. FASES DEL PROYECTO.

Para realizar el proyecto correctamente y de forma ordenada, debido a su dificultad, se ha dividido en cuatro fases. La primera fase corresponde a una etapa de aprendizaje, donde se adquieren los conocimientos necesarios tanto de ROS como de todo el software utilizado. La segunda fase ocupa la parte de configuración. Es una fase crítica pues es necesario configurar el hardware correctamente. Para ello hay que conocer muy bien el software y tener muy claros los conceptos básicos.

En la tercera fase se desarrolla el proyecto en sí. Teniendo todo estudiado y configurado es el momento de hacer que funcione. Esta etapa ocupará la mayor parte del proyecto.

La cuarta y última etapa comprende todas las pruebas del sistema desarrollado. Las pruebas se realizan tanto en el laboratorio como en el coche autónomo con el sistema ya instalado.

1.1. Fase 1: Aprendizaje.

Antes de comenzar el proyecto es necesario aprender y estudiar las herramientas que se van a utilizar. Es una etapa en la cual hay que dedicar tiempo y concentración pues es la base de todo el proyecto. Al principio del aprendizaje es normal sentirse desanimado pues parece que no aprendes nada. A medida que pasa el tiempo y te familiarizas tanto con ROS como con los drivers de Kinect se hace más llevadero. Por tanto el aprendizaje de estos conceptos se pueden asemejar a una curva exponencial.

Lo primero que hay que aprender es qué es ROS, como se usa y para que sirve. Hablaremos de nodos, topics o temas, suscriptores, publicadores… Para aprender bien todos estos conceptos es recomendable entrar en la web de ROS y realizar los tutoriales que viene en ella. Cuando ya estemos familiarizados con ROS, el siguiente paso es conocer la Kinect y sus drivers.

Cuando estemos familiarizados con ROS comenzaremos con la Kinect. Hay que saber que drivers usa y como utilizarlos. Sera necesario conocer las dependencias necesarias para que funcione la Kinect en el sistema.

Con la Kinect funcionando, hay que aprender a manejar las librerías PCL. Con ellas podremos trabajar sobre la imagen y los datos proporcionados por el sensor. Esta parte es fundamental para el proyecto pues el objetivo principal es el tratado de las imágenes obtenidas.

1.2. Fase 2: Configuración.

En esta parte se configura y se prepara tanto el software como el hardware que se usará en el proyecto.

Lo primero es instalar le versión 15.04 de Ubuntu en la placa de desarrollo Odroid. No será necesario instalar los driver de Kinect ni las librerías OpenCv ya que vienen instaladas previamente en la imagen de Ubuntu. En cambio, será necesario instalar las dependencias que hacen funcionar la kinect en ROS.

Esta versión de Ubuntu incorpora ROS Jade, por lo que solo será necesario preparar ROS para el proyecto. Sera necesario crear un entorno de trabajo y los nodos necesarios para el proyecto.

Para poder dar esta fase por terminada hay que conectar todo y comprobar que funciona correctamente.

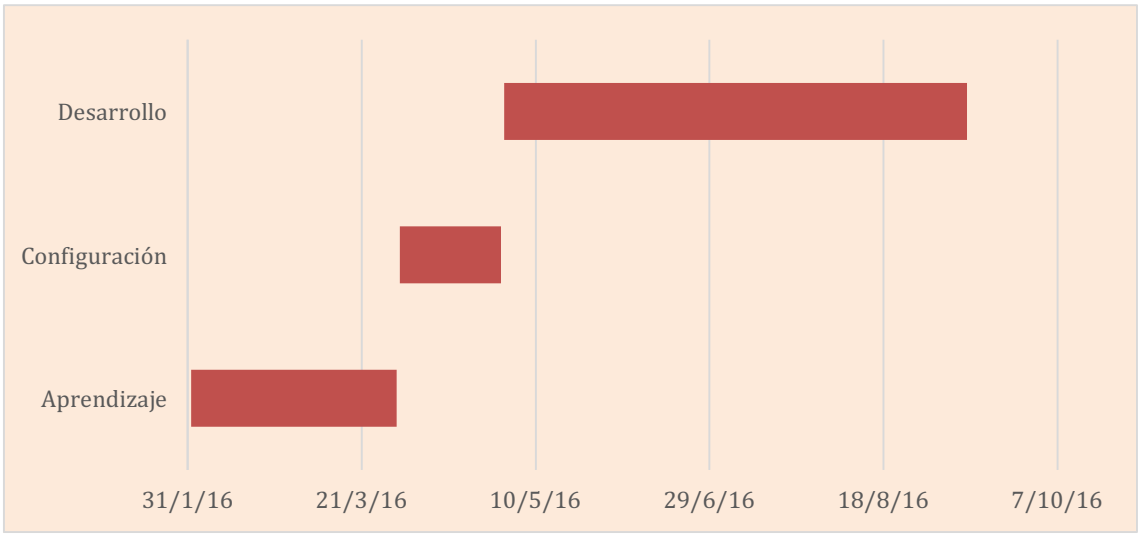
1.3. Fase 3: Desarrollo

Cuando esta todo configurado y funcionando, es el momento de dotar a nuestro proyecto de las funciones.

El objetivo del proyecto es crear un mapa de ocupación a partir de los obstáculos detectados con la Kinect. Se debe obtener una nube de puntos del sensor y con ella ir dibujando el mapa. Este mapa será publicado para, posteriormente ser utilizado por un coche autónomo.

TAREA	DESCRIPCIÓN
Fase 1	Corresponde al aprendizaje previo a la realización del proyecto
ROS	Sistema operativo utilizado
PCL	Herramientas para procesamiento de imagen
Fase 2	Corresponde a la configuración del sistema e instalación de herramientas necesarias
Ubuntu	Preparación e instalación de Ubuntu 14.04 en Odroid
ROS	Instalación ROS Jade
PCL	Instalación librerías PCL
Dependencias Kinect	Instalación IAI Kinect y software necesario
Fase 3	Corresponde al desarrollo práctica del proyecto
Paquete del programa	Preparación del sistema de archivos
Suscripción Kinect	Suscripción al nodo sd/points que proporciona los datos con los que se trabajará - nube de puntos
Detección del plano suelo	Algoritmo para detección del plano correspondiente al suelo
Eliminación puntos del suelo	Eliminación de la nube de los puntos que pertenezcan al plano suelo detectado
Calibración Kinect	Algoritmo para cálculo de la altura a la cual se coloca la Kinect
Eliminación de puntos innecesarios	Algoritmo para eliminar los puntos que estén fuera del intervalo definido
Transformación de la nube	Transformación de la nube filtrada anteriormente en función de los extrínsecos de Kinect
Publicación de la nube transformada	Publicar un topic en ROS con la nube transformada para permitir su uso en otros paquetes
PointCloud to LaserScan	Obtener un barrido laser a partir de la nube de puntos publicada
TOTAL PROYECTO	

Tabla 1: Fases del proyecto


Desarrollo
Configuración
Aprendizaje
31/1/16
21/3/16
10/5/16
29/6/16
18/8/16
7/10/16

CAPÍTULO 5:

APRENDIZAJE Y CONFIGURACIÓN

1. FASE 1: AUTOAPRENDIZAJE.

En esta fase se aprenden los conceptos necesarios para el desarrollo del proyecto. Es una fase necesaria para familiarizarse con el software y las librerías que se van a utilizar para conseguir los objetivos del proyecto.

A continuación se explicarán algunos de estos conceptos. En el anexo se adjuntan ejemplos que explican los términos descritos.

1.1. ROS

ROS es un meta-sistema operativo para robot. Es una librería de código que debe ser instalada en un sistema operativo y que se accederá a ella mediante un programa ejecutable. ROS se abstrae del hardware, controla los dispositivos a bajo nivel, implementa las funcionalidades más comunes en el manejo de robots, intercambia mensajes entre procesos y puede administrar paquetes de código a través de múltiples ordenadores.

ROS funciona con una estructura de red tipo p2p – puesto a puesto – donde los procesos se comunican entre sí usando los modos de comunicación de ROS. Cuando se utilizan servicios, la comunicación es síncrona. En cambio, cuando se usan topics es asíncrona.

Para programar en ROS se puede usar C++, Phyton o Lisp. En este proyecto usaremos C++ ya que se ha visto a lo largo de la carrera y nos proporciona más funcionalidades que los otros dos sistemas.

1.1.1. Sistema de ficheros ROS.

ROS se compone de:

- **Paquetes** – packages: un paquete puede contener nodos, librerías conjuntos de datos, archivos de configuración…
- **Manifiestos** – manifiests: son archivos .xml que proporcionan información sobre dependencias.
- **Pilas** – stacks: son colecciones de paquetes.
- **Manifiestos de Pila:** mismo concepto que los paquetes.

- **Tipos de mensajes:** son archivos con extensión .msg almacenados en *my_package/msg/Mimessagetype.msg.* Describen la estructura de los mensajes enviados por ROS.
- **Tipos de servicios:** son archivos con extensión .srv almacenadas en *my_package/srv/Myservicetype.srv.* Definen las peticiones y respuestas de las estructuras de datos para los servicios en ROS.

1.1.2. Elementos de computación.

- **Nodos:** son procesos ejecutables. Un nodos se programa usando una librería cliente ROS, por ejemplo roscpp o rospy. En este proyecto usaremos roscpp pues programaremos en C++.
- **Maestro:** asignan nombres a los elementos que conforman el modelo del robot. Sin el los nodos no se encuentran y por tanto no pueden comunicarse.
- **Servidor de parámetros:** forma parte del maestro y permite almacenar datos por clave en una localización centralizada.
- **Mensajes:** son estructuras de datos compuestas por campos tipificados.
- **Temas – topics:** son mensajes transportados mediante semántica de publicación/subscripción. Un nodo publica el topic y otro se suscribe a este topic para obtener datos.
- **Servicios:** es un sistema que atiende al modelo de petición/respuesta. Un nodo ofrece un servicio con un nombre determinado. Un nodo cliente usa este servicio enviando un mensaje de petición y esperando una respuesta.

1.1.3. Concepto de nodo

Los nodos son programas ejecutables que realizan funciones concretas, por ejemplo acceso a un motor, mapeo del entorno, planificación de rutas etc etc. La principal venta en la utilización de nodos es su diseño modular. Los nodos son compilados individualmente unos de otros, ejecutados y gestionados por un nodo principal (ROS Master). Los nodos se comunican entre sin mediante el uso de mensajes y topic con el fin de llevar a un objetivo en común. Al ser un diseño modular y ser cada nodo independiente nos permite afrontar los proyectos de una manera mucho más sencilla.

Cada nodo está escrito usando la librería cliente de ROS. La librería cliente de ROS es una colección de código que facilita el trabajo a los programadores en ROS. Mediante el uso de esta librería podemos no solo crear los nodos sino los también los publicadores y subscriptores de los mensajes y topics en los nodos, demás de los servicios y sus parámetros. Podemos utilizar esta librería tanto en C++ como en Phyton.

Cuando trabajamos con ROS podemos ver los nodos que se están ejecutando si introducimos en una terminal la siguiente sentencia:

```
$ rosnode list
```

1.1.4. Concepto de topic.

Es el modo de comunicación entre nodos en ROS. Cuando un nodo necesita publicar datos lo hace a través de un topic. Una vez publicado, si otro nodo necesita trabajar con los datos que otro ha publicado, debe suscribirse al topic.

Podemos hablar entonces de nodos publicadores – los que publican los topics – y de nodos suscriptores – los que se suscriben a los topics.

Para poder ver que topics hay disponibles para publicar y suscribirse ejecutamos en la terminal la siguiente sentencia:

```
$ rostopic list -v
```

1.2. Librerías PCL

Como ya se ha comentado, Point Cloud Librery viene instalada en la imagen de Ubuntu montada en Odroid. Vamos a aprender como crear proyectos usando esta librería y a visualizar nubes de puntos en Ubuntu.

Los proyectos que crearemos tendrán la siguiente estructura de carpetas.

- /Proyecto
  - CMakeLists.txt
  - /build
  - /src
    - Proyecto.cpp

El código del programa se encuentra en el archivo.cpp del proyecto mientras que las directivas para compilarlo se encuentran en el CMakeLists.txt. Una vez que hayamos programado lo que necesitemos en el .cpp es necesario compilar el proyecto.

Para ellos accedemos al directorio build del proyecto desde terminal y compilamos (cmake ..). Cuando termine de compilar montamos el proyecto (make). Por último lanzamos el ejecutable (./mi_proyecto).

En el anexo se puede ver un ejemplo en el que se crea y se rellena una nube de puntos. Seguidamente se le aplica un filtro y por último se visualiza.

2. FASE 2: CONFIGURACIÓN.

2.1. Instalar imagen Ubuntu en SD o eMMC.

Como ya se ha explicado, se montara una imagen de Ubuntu que ya incorpora ROS y los drivers de la Kinect. Esta imagen se puede descargar en el siguiente repositorio.

http://forum.odroid.com/viewtopic.php?f=95&t=16149

Una vez la tengamos descargada podemos elegir dos opciones: montarla en una micro SD de mini 8GB o montarla en una tarjeta eMMC. La segunda opción es más recomendable pues la eMMC es más rápida que una tarjeta SD convencional. Si elegimos esta opción necesitaremos un adaptador para conectarla al PC y montar la imagen.

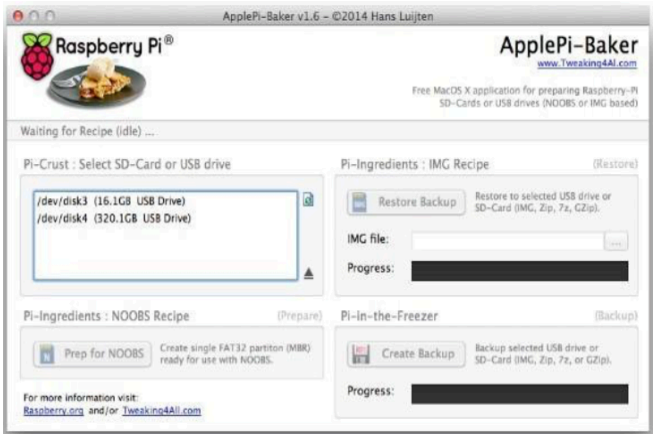


Ilustración 3: Instalación de Ubuntu 14.04 en SD con Apple Pi Baker

El proceso para montar la imagen de Ubuntu es el mismo en cualquiera de las dos opciones. Para hacerlo con un Mac es necesario tener instalado ApplePi Baker.

En la parte izquierda de la pantalla de ApplePi Baker vemos una lista de tarjetas SD o dispositivos USB. Hay que elegir el que corresponda a nuestra tarjeta SD o en su caso eMMC, según cual usemos. En la parte derecha elegimos el archivo que contiene la imagen de Ubuntu descargada. Una vez hecho esto grabamos la imagen.

Cuando termine, podremos desmontar la tarjeta del PC e introducirla en la placa Odroid para seguir trabajando.

2.2. Configuración Odroid.

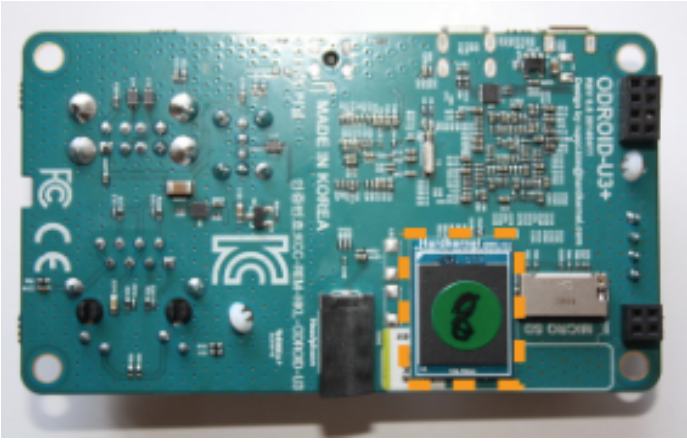


Ilustración 4: Situación de tarjeta eMMC en Odroid

Es el momento de conectar nuestra tarjeta con la imagen de Ubuntu a la placa Odroid. Según hayamos elegido SD o eMMC debemos conectarlas en un sitio o en otro.

El lugar de conexión de la eMMC viene recuadrado en naranja. Si elegimos una tarjeta SD, el lector de tarjetas se encuentra en la parte delantera de la placa, junto a la toma de corriente.

Llegado a este punto es necesario conectar los periféricos para poder trabajar más cómodamente. Necesitaremos un monitor con entrada HDMI y un cable HDMI-micro HDMI. Conectaremos también un teclado y un ratón dejando libre el puerto USB 3.0 que estará destinado para la Kinect – recordemos que la Kinect 2 solo funciona con un USB 3.0. Finalmente conectamos la placa a corriente.

Es el momento de instalar las dependencias necesarias para poder usar la Kinect. Consultando el anexo se puede acceder a un repositorio donde descargar un paquete con todas las dependencias. Una vez descargado y copiado en Odroid, es necesario compilarlas.

La fase de configuración se puede dar por concluida. Ante cualquier imprevisto, se crea una imagen del sistema operativo. Esto evitará tener que configurar todo de nuevo si ocurre algún problema en la fase de desarrollo.

CAPÍTULO 6:

DESARROLLO

1. FASE 3: DESARROLLO

Para comenzar esta fase es necesario saber como funciona el hardware y el software. El punto de partida es tener la Kinect conectada a la placa Odroid, pero, ¿cómo se obtienen las imágenes que proporciona la Kinect?

El sensor Kinect se conecta Ubuntu/Linux usando sus drivers – libfreenect2. Una vez que está conectada correctamente al sistema operativo hay que conectarla con ROS. Para ello usamos el puente IAI Kinect 2. Es un nodo de ROS que va a publicar diferentes topics con información tanto de la cámara como de las imágenes que ésta obtiene.

Por otra lado, hay que realizar el programa para el procesamiento de la imagen. Se va a trabajar con nubes de puntos, por ello hay que suscribirse al topic de IAI Kinect2 que da la nube de puntos.

Cuando este todo listo será el momento de ejecutarlo y probar su funcionamiento. Esta parte se explicará más adelante.

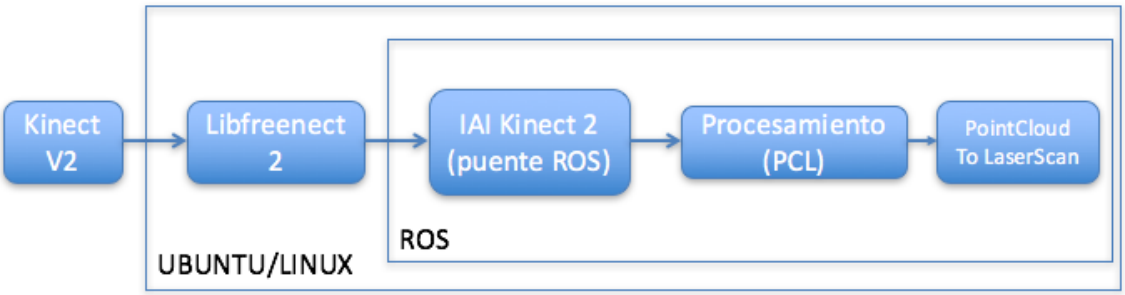


Ilustración 5: Diagrama del software

Este programa debe realizar el procesamiento de la nube de puntos usando las librerías PCL. Para conseguir esta nube debe suscribirse al topic points de IAI Kinect2. La nube se obtiene y se guarda en un objeto de tipo Point Cloud. Se realiza el procesamiento y se publica la nube procesada. Por tanto este nodo tiene tres funciones:

- Suscriptor.
- Procesamiento con PCL.
- Publicador.

1.1. Crear el paquete del programa.

Para que ros pueda ver y ejecutar el programa se debe crear un paquete que lo contenga dentro del workspace. Desde terminal se accede a la carpeta /src del workspace. Aquí se crea el directorio en el cual se ubicara el programa.

```
$ cd ~/catkin_ws/src
$ roscreate-pkg Kinect_cloud std_msgs rospy roscpp
```

Para comprobar que el paquete ha sido creado correctamente y ROS puede verlo se ejecutan el siguiente comando.

```
$ rospack find procesamiento
```

Si se encuentra el paquete, se mostrará lo siguiente en la terminal.

```
/home/rssierra/catkin_ws/src/kinect_cloud
```

Con el paquete correctamente creado, se accede a la carpeta src del proyecto. En ella se crea un archivo .cpp que contendrá el código del programa.

```
$ gedit Kinect_cloud.cpp
```

Si todo ha ido bien, la estructura de carpetas debe ser la mostrada a continuación.

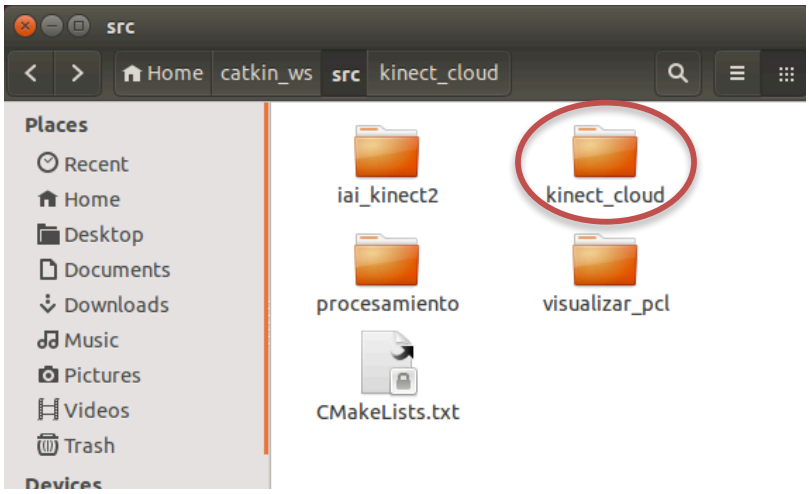


Ilustración 6: Contenido del paquete Kienct_Cloud en el workspace de ROS

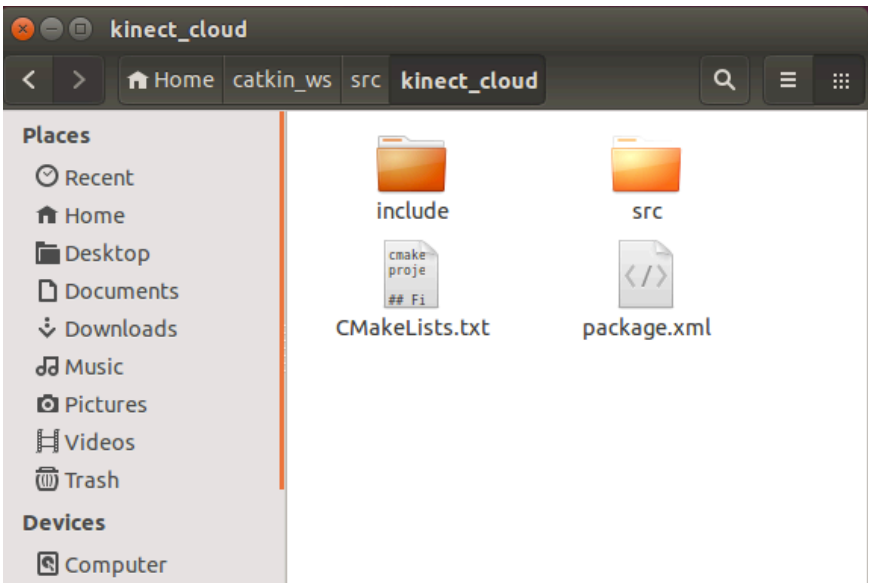


Ilustración 7: Contenido del paquete Kinect_Cloud

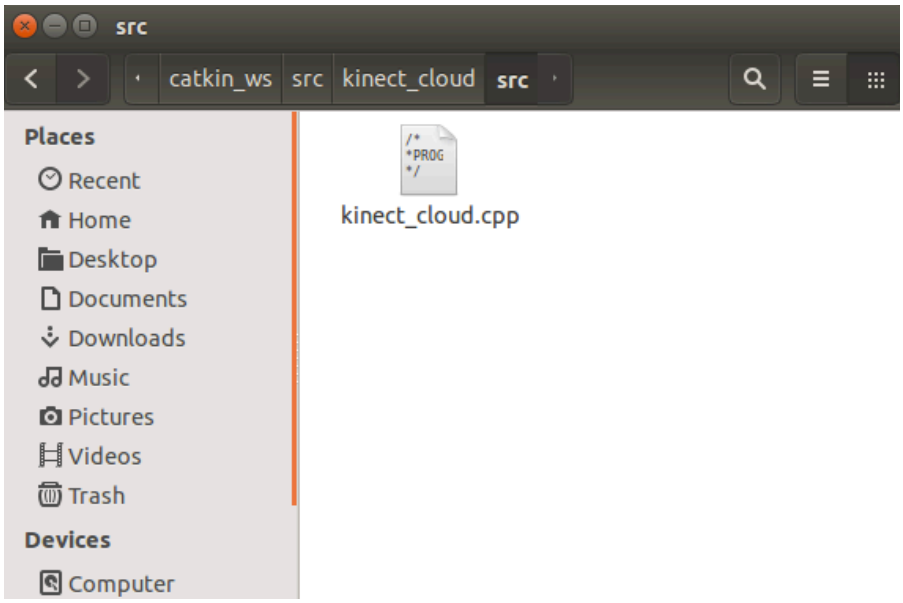


Ilustración 8: Ubicación del archivo .cpp

1.2. Descripción del programa.

El programa desarrollado tiene la función de obtener una nube de puntos de tipo PointCloud, procesarla y publicarla en el sistema ROS. Con ello se permite que los resultados obtenidos por el programa se puedan utilizar en otros paquetes en ROS.

Para realizar esta tarea, se obtiene la nube de puntos directamente de la Kinect a través del puente IAI_Kinect2. Una vez hecho esto, se procesa. El procesamiento de esta nube consiste en el filtrado según ciertos parámetros del sensor. Una vez filtrada correctamente, se publica la nube final en ROS.

Esta nube será usada por un tercer paquete para transformarla en un barrido laser con el cual se obtiene un mapa 2D. Este mapa puede ser usado para diferentes aplicaciones.

1.2.1. Librerías necesarias

En este apartado se detallan las librerías a incluir para poder codificar y compliar el programa correctamente.

1.2.1.1. Librerias de ROS

En primer lugar es necesario incluir las librerías de ROS para poder usar todas las funciones que este software tiene disponibles, como subcriptores, publicadores, mensajes, etc.

Para ello se incluye en el código la siguiente bibloteca: **ros/ros.h**.

1.2.1.2. Mensajes de tipo PointCloud

A la hora de obtener la nube de puntos de la Kinect, ésta se guarda en un mensaje de ROS de tipo PointCloud2. Para poder utilizar este tipo de mensajes se debe incluir la siguiente biblioteca: **sensor_msgs/PointCloud2.h**.

1.2.1.3. Librería PCL

Para poder usar nubes de puntos se deben incluir dichas librerías, que han sido instaladas previamente. Esta librería es **pcl/point_cloud.h**. Aunque se incluya esta librería, serán necesarias algunas librerías mas de PCL para poder realizar diferentes operaciones con nubes de puntos.

1.2.1.4. Uso nube de puntos y ROS

Para poder trabajar con nubes de puntos en ros sin ningún problema, es necesario incluir las siguiente librería: **pcl_ros/point_cloud.h**.

1.2.1.5. Nubes de puntos de tipo PointCloud2

Existen diferentes tipos de nubes de puntos. La nube obtenida de Kinect es de tipo PCLPointCloud2. Para poder usar este tipo de datos en el programa es necesario incluir la siguiente biblioteca: **pcl/PCLPointCloud2.h**.

Ademas de este tipo de dato, es necesario usar nubes de tipo PointCloud.

1.2.1.6. Conversiones entre nubes de puntos

A lo largo de la ejecución del programa, son necesarias algunas conversiones entre tipos de nube de puntos. Para poder realizar estas conversiones de manera sencilla, ya que están implementadas dentro de las librecias PCL, es necesario incluir las siguientes librerías: **plc/conversions.h** y **pcl_conversions/pcl:conversions.h**.

1.2.1.7. Visualización de nube de puntos

Dentro de las librerías PCL, existe una herramienta que permite visualizar la nube de puntos. Esta herramienta es CloudViewer. Resulta de gran utilizad en el proyecto pues permite ver el resultado de las diferentes transformaciones realizadas. Para poder usarla es necesario incluir la siguiente librería: **pcl/visualization/cloud_viewer.h**.

1.2.1.8. Utilización del método RANSAC y modelos.

En el Anexo II se encuentra definido el método RANSAC y las ventajas que aporta al proyecto. Dentro de este método, se encuentran definido una serie de modelos geométricos que son muy importantes en el programa descrito. El modelo usado es el de plano paralelo – parallel plane. Se usará para obtener la definición del plano del suelo, explicado más adelante.

Para poder usar estas herramientas es necesario incluir las siguientes librerías:

pcl/simple_consensus/method_types.h

pcl/simple_consensus/model_types.h

pcl/simple_consensus/sac_model_parallel_plane.h

pcl/simple_consensus/ransac.h

pcl/segmentation/sac_segmentation.h

1.2.1.9. Filtros PCL

A lo largo de la programación, ha sido necesario usar diferentes filtros para adecuar la nube de puntos a lo que requerían las diferentes funciones del programa. Se han aplicado filtros, por ejemplo, para eliminar ciertos puntos que no eran necesarios. Para poder usar estos filtros, hay que incluir las siguientes librerías:

pcl/filters/voxel_grid.h

pcl/filters/passthrough.h

pcl/filters/extract_indices.h

1.2.1.10. Transformación de nube de puntos

Para hacer que el programa funcione correctamente, independientemente de la posición de la cámara, ha sido necesario realizar transformaciones a la nube de puntos obtenida en función de los parámetros extrínsecos de la Kinect. Estas transformaciones están incluidas en la siguiente librería: **pcl/common/transform.h**.

1.2.1.11. Uso de mensajes de tipo LaserScan

Como ya se introdujo al inicio del proyecto, la función del programa es crear un escaneo laser a partir de las imágenes que obtiene de Kinect. Esta salida será por tanto de tipo LaserScan. Para poder publicarla en un mensaje de ROS es necesario incluir la siguiente librería: **sensor_msgs/LaserScan.h**.

1.2.1.12. Librerias C++

Para poder usar algunas funciones que proporciona C++ hay que incluir algunas librerías del propio lenguaje. Se usan funciones matemáticas – senos, cosenos, raíces cuadradas, etc – así como las funciones para imprimir datos por pantalla. También se usan, aunque en contadas ocasiones, funciones temporales. Para poder usar todas ellas, es necesario incluir las siguientes librerías:

iostream

ctime

cmath o **math.h**

1.2.2. Función main

Como en todos los programas informáticos, existe una función principal o main. En este caso el main corresponde a la función principal del nodo, por lo que es la que se ejecute en primer lugar cuando se llame al nodo.

En primer lugar si inicia el nodo en ros con la sentencia **ros::init(arc,argv,"kinect_cloud")**. Siendo 'kinect_cloud' el nombre del nodo. Seguidamente, se declaran los publicadores y subscriptores. En este caso tenemos un subscritor que será el encargado de obtener la nube de puntos de Kinect. por su parte, los publicadores serán los encargados de publicar, mediante mensajes de ROS, las salidas del programa. En este caso tenemos tres: el primero publica la nube en formato PCLXYZ, el segundo publica la nube bajo un mensaje de ROS y el tercero publica el escaneo laser. A continuación se muestra la declaración de subscriptores y publicadores:

- **ros::Subscriber sub = n.subscribe("/kinect2/sd/points",10,Callback)**

  Para obtener la nube de puntos hay que subcribirse al topic /kinect2/sd/points. Se especifica el tamaño de cola que serán 10 frames. El tipo de nube que se obtiene será de tipo sensor_msgs::PointCloud2.

- **pub_cloud = n.advertise<pcl::PointCloud<pcl::PointXYZ> > ("/kinect_cloud/kinect2_cloud", 10)**

  En esta sentencia se declara un publicador bajo el cual se publicara la nube de puntos de tipo pcl::PointXYZ. Se especifica el nombre '/Kinect_cloud/kinect2_cloud' y un tamaño de cola de 10 frames

- **pub_cloud_msg = n.advertise<sensor_msgs::PointCloud2>("/kinect_cloud/kinect2_msg", 10)**

  En este caso se publica la misma nube de puntos pero bajo un mensaje de ros de tipo sensor_msgs::PointCloud2. El topic recibe '/kinect_cloud/kinect2_msgs' como nombre y un tamaño de cola de 10 frames.

- **pub_laser = n.advertise<sensor_msgs::LaserScan>("/kinect_cloud/laserscan", 10)**

El escaneo laser obtenido tras la ejecución completa del programa se publica bajo un mensaje de ros de tipo LaserScan. Éste recibe el nombre '/kinect_cloud/laserscan' y tiene un tamaño de cola de 10 frames.

El siguiente proceso que se realiza es asignar a la matriz de transformación y rotación los coeficientes en función de la posición de la cámara. En este caso se ha elegido la matriz identidad debido a la colocación de la misma. Al definir los coeficientes de la matriz como variables globales, si cambiara la posición de la cámara, solo sería necesario cambiar estas variables y cambiaría la matriz de transformación.

Por último se declaran los parámetros de la segmentación, que se usaran para obtener el plano del suelo.

seg.setModelType (pcl::SACMODEL_PARALLEL_PLANE);

En esta primera sentencia se especifica el tipo de modelo con el cual se va a trabajar. Como el objetivo es sacar un modelo que defina el plano del suelo, se elige un modelo para un plano paralelo.

seg.setEpsAngle (conver_rad(20));

En la segunda sentencia se indica el ángulo que, aproximadamente, formara el suelo con la cámara. Según la posición en la cual se ha colocado la Kinect, se estima que se formen 20º entre la cámara y el suelo.

seg.setMethodType (pcl::SAC_RANSAC);

Seguidamente se especifica con que método se realizará la segmentación. En este casos e usa el método RANSAC, explicado en el Anexo II.

seg.setDistanceThreshold (0.1);

Por último se indica la distancia limite, es decir, una distancia máxima a la que se espera que haya puntos pertenecientes al suelo. Se ha elegido una distancia de 10cm. Es decir, se considera que un punto pertenece al plano del suelo si está a menos de 10cm del mismo.

1.2.3. Función Callback del nodo

La función Callback pertenece al topic **/kinect2/sd/points**. Cada vez que llega un nuevo mensaje a este topic se ejecuta la función Callback. En ella se encuentra el código necesario para general el escaneo laser que será publicado.

En primer lugar se realizan las transformaciones de la nube de puntos a un tipo de dato con el cual se pueda trabajar cómodamente. La nube es publicada en el topic bajo un mensaje de ROS de tipo **sensor_msgs::PointCloud2.**

La primera conversión pasa la nube del mensaje a un tipo de dato propio de librerías PCL. El producto de esta conversión es una nube de tipo **pcl::PCLPointCloud2**. Para aplicar esta conversión basta con usar la siguiente función:

```
void pcl_conversions::toPCL (const sensor_msgs::PointCloud2 input,
                             pcl::PointCloud2 output)
```

Al realizar la llamada, se deben pasar como argumentos la nube de puntos original, es decir, la que proviene del mensaje de ROS y la nube de puntos de salida, es decir, la de tipo PCL.

Con la nube de salida, de tipo pcl::PCLPointCloud2, se realizara una segunda transformación para convertirla a una nube en la que los puntos estén definidos por sus coordenadas XYZ. El tipo de nube necesario es pcl::PointCloud<pcl::PointXYZ>. Esta nube XYZ debe ser declarada como puntero. La función necesaria para realizar la conversión es la siguiente:

```
pcl::fromPCLPointCloud2 (pcl::PointCloud2 input,
                         pcl::PointCloud<pcl::PointXYZ> output);
```

Una vez se ha convertido la nube al tipo de dato necesario, se llevan a cabo las operaciones que generarán el escaneo laser. Estas operación serán explicadas en los siguientes puntos. En primer lugar se realiza la detección del suelo y la eliminación de los puntos pertenecientes al mismo de la nube. Seguidamente se trasforma la nube en función de la posición de la cámara. A este proceso se le ha denominado calibración. A continuación se transforma la nube en función de los extrínsecos de la cámara. En este punto se obtiene la nube con la que se generara el escaneo laser, es decir, el mapa 2D.

Esta nube se publica en ROS con tipo pcl::PointCloud2 en el topic /kinect_cloud/kinect2_cloud. También se publica en un mensaje de ROS de tipo sensor_msgs::PointCloud2 en el topic /kinect_cloud/kinect2_msg. para poder publicar la nube en un mensaje de ROS hay que realizar la conversión a este tipo de dato. La conversión que hay que usar es la siguiente:

```
pcl::toROSMsg(pcl::PointCloud2 <pcl::PointXYZ>, sensor_msgs::PointCloud2);
```

Una vez publicada la nube se ejecuta la función **PCL_to_laserscan** para obtener el escaneo laser. El código de esta función, explicado más adelante, se ha obtenido del nodo **point_cloud_to_laserscan**. Por tanto, realiza la misma función que este nodo pero su ejecución es más sencilla.

Una vez se ha obtenido el escaneo laser, que corresponde al mapa 2D, se publica en ROS y se termina la ejecución del programa.

1.2.4. Detección y eliminación del plano suelo

Esta es el primer problema que hay que solucionar. Para poder obtener correctamente el plano 2D, es necesario eliminar los puntos de la nube que pertenecen al plano del suelo.

Para detectar este plano, se usa el método RANSAC. En este proyecto se trabaja con el modelo de plano paralelo – SACMODEL_PARALLEL_PLANE. Esto es debido a que el sensor Kinect está orientado paralelo al suelo. Se podría haber elegido para la detección un modelo perpendicular pero no resultaba tan preciso en la práctica.

Se crea un objeto de tipo SACSegmentation con el cual se obtendrá el modelo del plano deseado. Se especifica que debe ser paralelo a la cámara. Para asegurar que se trata del plano del suelo, se define un ángulo de 20 grados. Además, para ajustar la detección de puntos a este plano se especifica una distancia de 10cm.

Según estas especificaciones, se considera que el plano suelo será el plano paralelo al sensor y que además forme un ángulo de 10 grados con los ejes del sensor. En cuando a los puntos, se considera que un punto pertenece a este plano si se encuentra a una distancia igual o inferior a 10cm.

En el Anexo III se pueden ver las capturas de las diferentes pruebas de detección del suelo. Analizando estas imágenes, se puede decir que se detecta el suelo de forma satisfactorio y por tanto se puede pasar a eliminar los puntos que pertenecen al plano definido.

Una vez que se ha detectado el plano del suelo, es necesario filtrar la nube y eliminar los puntos pertenecientes a dicho plano. Para ello se utilizará el filtro 'extract' incluido en la librería PCL.

Se debe pasar a este filtro la nube capturada con el sensor Kinect y los puntos – inliers – detectados en el apartado anterior. De esta manera se pueden eliminar estos puntos de la nube.

1.2.5. Calibración Kinect

Una vez que se ha detectado el plano del suelo, es necesario calibrar la cámara en función de éste plano. En el proceso de calibración el objetivo es obtener la altura a la cual se encuentra la cámara del suelo.

Como se ha obtenido el modelo del plano, se tiene la ecuación normal del plano, del tipo:

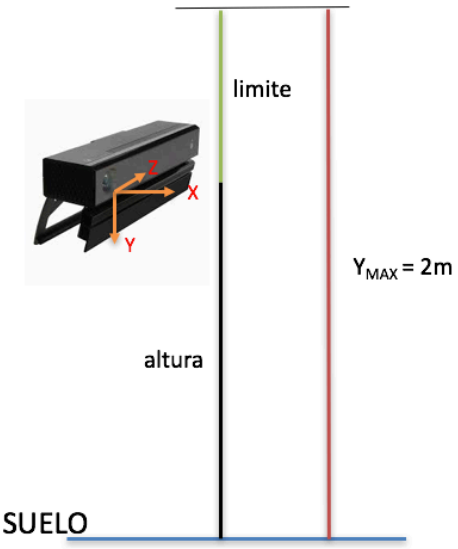
$$Ax + By + Cz + D = 0$$

Donde los coeficientes A, B, C y D se han obtenido durante la detección del suelo.

Tomando como origen de coordenadas la propia Kinect, el problema se resuelve calculando la distancia mínima desde el origen de coordenadas al plano. Esta distancia mínima corresponde con la perpendicular desde el origen al plano que define el suelo. Por tanto, esta distancia será la altura a la cual está ubicada la Kinect. Para ello, se aplica la siguiente ecuación:

$$altura = d(O, \pi) = \frac{|A \cdot x_o + B \cdot y_o + C \cdot z_o + D|}{\sqrt{A^2 + B^2 + C^2}}$$

Donde X_o, Y_o y Z_o son las coordenadas del punto al cual se calcula la distancia. En este caso será el origen, siendo sus coordenadas (0,0,0).



$$I = (\text{limite\_1}, \text{limite\_2})$$

limite_1 = -(Y_MAX - altura + Seg);
limite_2 = altura + Seg;

$$Seg = 0{,}1m$$

Ilustración 9: Calculo intervalo de puntos que se van a utilizar

Una vez se ha obtenido la altura de la Kinect, se puede calcular el intervalo de puntos con el cual trabajar. Está definido por dos limites, en función de una altura máxima de 2m y la altura de la Kinect. El objetivo de este intervalo es filtrar la nube para, de esta manera, trabajar únicamente con los puntos pertenecientes a este intervalo.

1.2.6. Filtrar la nube en función del intervalo calculado

Una vez se ha obtenido el intervalo de puntos con el cual se debe trabajar, es necesario eliminar todos los puntos de la nube que no pertenezcan a este intervalo. Esto se hace con el fin de reducir el tamaño de la nube para que la transformación en barrido laser sea más rápida.

Para filtrar la nube según el intervalo dado se usa el filtro PassTrough, incluido en las librerías PCL. Como *InputCloud* se pasa la nube con la cual se está trabajando. El parámetro *FilterFieldName* es el eje en el cual se va a aplicar el filtro. Como el intervalo esta cogido en el eje y, este deberá ser el eje en el cual se filtre la nube. A continuación, se pasan los límites del filtro, guardados en el parámetro *FilterLimits*. Los limites coinciden con los del intervalo calculado. Por último, se indica cual será la nube de salida. En este caso es la misma que la nube de entrada.

1.2.7. Transformación de la nube

En función de donde se coloque la Kinect tendrá un sistema de coordenadas u otro. Éste vendrá definido por los valores extrínsecos de translación y rotación. Se requiere por tanto transformar el sistema de coordenadas de la Kinect a un sistema de coordenadas real, que corresponde con la posición en la cual es instalada.

Para realizar esta transformación se definen dos matrices, una de translación de dimensiones 3x1, y una matriz de rotación de dimensiones 3x3.

$$R = \begin{bmatrix} \cos\beta & \sin\beta & 0 \\ -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} T_X \\ T_Y \\ T_Z \end{bmatrix}$$

Una vez definidas las matrices de rotación y translación, se puede aplicar la siguiente ecuación para transformar el sistema de coordenadas.

$$\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + T$$

Habiendo entendido como se transforman sistemas de coordenadas, se puede pasar este desarrollo a lenguaje informático y realizar la programación necesaria.

Para realizar la transformación se usa la función *transformPointCloud* incluida en las librerías PCL. Esta función recibe los siguientes parámetros:

- **Nube de puntos a transformar:** es la nube de puntos original, es decir, la que se va a transformar. En este proyecto se corresponde con la nube de puntos final, una vez filtrada según el intervalo de puntos obtenido.

```
const pcl::PointCloud< PointT > & cloud_in
```

- **Nube de puntos transformada:** es la nube de puntos transformada. En este proyecto el resultado de la transformación se guarda sobre la nube anterior.

```
const pcl::PointCloud< PointT > & cloud_out
```

- **Matriz de transformación:** se trata de una matriz de dimensión 4x4. Para declarar esta matriz es necesario usar la librería Eigen y definir una matriz de tipo Matrix4f.

  const Eigen::Matrix<Scalar, 4, 4> & transform

  Este parámetro, como se puede ver, requiere una matriz 4x4. Si se aplica la transformación descrita anteriormente, el resultado sería una matriz de 3x3. Por ello se define una matriz adicional en la cual se incluyen las matrices R y T. Incluyendo una fila adicional, se obtiene la matriz necesaria.

1.2.8. Publicar la nube

Como se explicó anteriormente, la nube se publica de tres formas. La primera publicación se realiza con la nube de puntos de tipo **pcl::PointCloud < pcl::PointXYZ >**. La segunda publicación se realiza con la nube de puntos de tipo **sensor_msgs::PointCloud2**, es decir, se publica la nube bajo un mensaje de ROS. Para ello se realiza la transformación de **pcl::PointCloud** a sensor_msgs::PointCloud2. Para ello se usa la función pcl::toROSMsg.

Por último se publica el escaneo laser. Éste se genera transformando la nube en **sensor_msgs::Laserscan**. De ello se encarga la función **PCL_to_laserscan**, cuyo código se ha obtenido del nodo **pointcloud_to_laserscan**.

1.2.9. Generar escaneo laser: función PCL_to_laserscan

El nodo **pointcloud_to_laserscan** se encarga de generar un escaneo laser a partir de una nube de puntos recibida por parámetro. Esta nube es recibida desde ROS a través de un mensaje de tipo **sensor_msgs::PointCloud2**. En este proyecto se ha obviado la ejecución de este nodo y, por tanto, el envío del mansaje de ROS. La solución ha sido coger el programa que usa el nodo y pasarle la nube en un tipo de dato **sensor_msgs::PointCloud2**, lo que ha hecho necesario la transformación de pcl a sensor_msgs descrita en apartados anteriores. A pesar de ello, se simplifica la ejecución del programa ya que requiere un nodo menos en ejecución, aumentando la rapidez de procesamiento.

La llamada a la función se realiza una vez publicada la nube como pcl y como mensaje en ROS. En el proyecto, esta función recibe el nombre de **PCL_to_laserscan**. La sentencia implementada para llamar a esta función sería la

siguiente: **PCL_to_laserscan(cloud_msg)**, donde cloud_msg es la nube de puntos de tipo sensor_msgs::PointCloud2.

Una vez en la función, el primer paso es construir la salida Laserscan que se publicará al final de la ejecución. Para ello se declara un objeto de tipo **sensor_msgs::LaserScan** al que habrá que especificarle diferentes parámetros.

sensor_msgs::LaserScan output;

Los parámetros que hay que especificar, una vez se ha declarado el objeto, son los siguientes:

- **Nombre**: que recibirá el frame en el cual se guardará el escaneo laser. se trata de una variable tipo string. Como en este proyecto no es necesario, pues se publicará en un topic bajo un mensaje de ROS, se puede dejar en blanco.

  output.header.frame_id = "";
- **Ángulo mínimo:** especifica el ángulo mínimo del intervalo de barrido. Es decir, el escaneo laser se realiza para un intervalo de la nube de puntos. Este intervalo viene definido por un ángulo. En este caso, el ángulo mínimo marca el valor más pequeño que tomará el ángulo de barrido.

  output.angle_min = angle_min_;

  El ángulo mínimo se define por una variable global cuyo valor es de -55º.
- **Ángulo máximo:** especifica el ángulo máximo del intervalo de barrido. En este caso, el ángulo máximo marca el valor más grande que tomará el ángulo de barrido.

  output.angle_max = angle_max_;

  El ángulo máximo se define por una variable global cuyo valor es de 55º.

Se define por tanto el siguiente intervalo de barrido, en grados: (-55,55). Debe ser así pues se toma 0º como el ángulo en el cual se encuentra el origen de coordenadas de la cámara.

- **Incremento angular:** especifica la resolución del escaneo laser, es decir, como varía el ángulo de barrido. Para este proyecto se define este

incremento mediante una variable global que toma el valor de 0.087º. Se aplica este incremento debido a las características de la nube. Tras la realización de diferentes pruebas y posteriores ajustes de este incremento se llegó a la conclusión de que si el ángulo varía en 0,087º el escaneo laser resulta más fiable.

output.angle_increment = angle_increment_;

- **Tiempo de escaneo:** especifica el tiempo que dura el escaneo. Se define mediante una variable global con valor de 0,033ms. Es decir, el tiempo invertido en el escaneo es de 0,033ms. Este valor no se ha modificado respecto al código original pues no afectaba ni a la ejecución del programa ni al escaneo generado.

  output.scan_time = scan_time_;
- **Rango máximo:** especifica el rango máximo devuelto en metros. Es decir, es la mayor medida registrada en el escaneo laser. Se define mediante una variable global cuyo valor es de 10m.

  output.range_max = range_max_;
- **Rango mínimo:** especifica el rango mínimo devuelto en metros. Es decir, es la menor medida registrada en el escaneo laser. Se define mediante una variable global cuyo valor es de 0,05m.

  output.range_min = range_min_;

Según estos dos últimos parámetros la menor distancia representada en el escaneo laser es de 0,05m, mientras que la máxima es de 10m. Esto quiere decir que no se representa ningún punto por debajo del rango mínimo ni por encima del rango máximo.

Una vez que se han especificado los parámetros que definen el escaneo laser, es necesario saber cuántos rayos hay que crear. Se entiende por rayos las diferentes líneas de escaneo. Se obtienen las distancias para todo punto que pertenezca a cada una de estas líneas.

El número de rayos o líneas de escaneo depende del ángulo máximo, del ángulo mínimo y de la resolución, es decir, del incremento angular. Para obtener el número de líneas se aplica la siguiente ecuación.

$$num\ ranges = \frac{angulo\ \max - angulo\ min}{incremento\ angular}$$

El número de líneas equivale al entero inmediatamente superior al resultado de la ecuación anterior. Esta operación se codifica de la siguiente manera:

```
double num_ranges = ((output.angle_max - output.angle_min))/(output.angle_increment);
uint32_t ranges_size = std::ceil(num_ranges);
```

Este número de líneas se guarda en un vector. Por tanto es necesario redimensionar este vector para que su dimensión corresponde con el número de líneas calculado. Para ello se codifica la siguiente sentencia.

output.ranges.resize(ranges_size);

Para garantizar que este vector está vacío, es decir, todos sus elementos son cero, se recorre mediante una sentencia for. Se da a cada elemento el valor del rango máximo, es decir, 10m. De esta manera, si no se ha detectado ningún objeto, se representara una línea continua a 10m de la cámara.

Llegando a este punto, se puede proceder a generar el escaneo laser. Para ello es necesario iterar la nube de puntos recibida por la función PCL_to_laserscan. Esta iteración se realiza en los tres ejes de coordenadas – X, Y, Z - mediante un bucle for. De esta forma se recorren todos los puntos de la nube y se procederá a su análisis.

Para evaluar cada punto, se evalúa primero que éste tenga datos, es decir, que no sea un punto vacío – erróneo. Seguidamente se evalúa que este dentro del intervalo de altura y dentro del rango de escaneo. Si todo es correcto se guarda en el vector de rangos para posteriormente publicarlo una vez concluida la rutina. Todos los filtros o evaluaciones mencionadas se realizan mediante una sentencia if. Se evalúa el caso desfavorable por tanto, si éste se cumple, la iteración continuará con el siguiente punto.

En primer lugar, se evalúa que el punto cogido no este vacío. Para ello se realiza la siguiente sentencia if:

```
if (std::isnan(*iter_x) || std::isnan(*iter_y) || std::isnan(*iter_z))
{
  std::cout<<"rejected for nan in point( " << *iter_x << ", "
                                           << *iter_y << ", "
                                           << *iter_z << ")" << endl;
  continue;
}
```

Si cualquier coordenada del punto es nula se cumple la sentencia if y el programa muestra por pantalla el error. Seguidamente continua el bucle for para evaluar el siguiente punto.

El siguiente paso es comprobar que el punto está dentro del rango de altura. Este intervalo se calculó en el punto *7.2.5. Calibracion de Kinect.* Esta comprobación se realiza mediante la siguiente sentencia if:

```
    double h = 0;
    if(*iter_y < 0) h = -(*iter_y);
    if(*iter_y > 0) h = (*iter_y);
if (h > max_height_ || h < min_height_)
{
   std::cout<<"rejected for height " << *iter_y << " not in range ( " << min_height_ << ", "
                                                                     << max_height_ << ")" << endl;
  continue;
}
```

Se define una variable de tipo double y mediante los dos primero if se le da el valor positivo de la coordenada Y del punto. Esto es necesario debido a la orientación de los ejes de coordenadas de la Kinect, girados 180º respecto a los de laserscan. Después de hacer esta transformación, se evalúa que la coordenada Y este dentro del intervalo calculado en el punto *7.2.5.* Si no fuera así, se muestra el error por pantalla y se procede a evaluar el siguiente punto.

Seguidamente, se comprueba que el rango. Esta variable, de tipo double, se calcula realizando la hipotenusa entre la coordenada X y Z del punto. Solo se comprueba que el rango supere el valor del rango mínimo establecido. No es necesario comprobar el rango máximo ya que si el punto no cumple la condición anterior se daría por erróneo y en el escaneo laser se representaría con el rango máximo, es decir, 10m. Para realizar esta comprobación se realiza la siguiente sentencia if:

```
double range = hypot(*iter_x, *iter_z);
if (range < range_min_)
{
   std::cout<<"rejected for range " << range << " below minimum value " << range_min_ << " Point: ( "
                                                                        << *iter_x << ", "
                                                                        << *iter_y << ", "
                                                                        << *iter_x << ")" << endl;

  continue;
}
```

En el caso de que el rango calculado sea inferior al rango mínimo, se muestra el mansaje de error por pantalla y se evaluará el siguiente punto.

Para terminar, se evalúa que el punto se encuentra en un ángulo que pertenezca al intervalo comprendido entre el ángulo mínimo y el máximo. Para ello se calcula el ángulo del punto mediante la arco tangente cuadrada del cociente entre la coordenada Y por Z. La sentencia if resultante es:

```
double angle = atan2(*iter_x, *iter_z);
if (angle < output.angle_min || angle > output.angle_max)
{
   std::cout<<"rejected for angle " << angle << " not in range ( " << output.angle_min << ", "
                                                                  << output.angle_max << ")" << endl;

  continue;
}
```

Como en las condiciones anteriores, si el ángulo no es correcto se informa por pantalla del error y se procede a evaluar el siguiente punto.

El punto es correcto si ninguna condición se ha cumplido. En este caso se procede a guardar su información en el vector de rangos. Para realizar esta tarea es necesario calcular a que índice pertenece y su rango, que ya se calculó anteriormente.

El índice depende del ángulo calculado en la comprobación correspondiente, del ángulo mínimo y del incremento angular. Se calcula aplicando la siguiente fórmula:

$$indice = \frac{angulo - angulo\ minimo}{incremento\ angular}$$

A continuación se evalúa si el rango es menor que el rango máximo. En este caso, se guarda en el vector. En caso contrario se dejara este índice como rango máximo.

El código desarrollado para guardar los datos en el vector es el siguiente:

```
//overwrite range at laserscan ray if new range is smaller
int index = (angle - output.angle_min) / output.angle_increment;
if (index < 0)
{
      index = - index;
}
if (range < output.ranges[index])
{
  output.ranges[index] = range;
}
```

Para concluir con la ejecución del programa se publica el escaneo laser de la siguiente manera:

```
pub_laser.publish(output);
```

Tras esta línea de código finaliza el programa. Se volverá a ejecutar cuando se publique un nuevo mensaje en el topic **"/kinect2/sd/points"**, es decir, cuando llegue una nueva nube de puntos.

CAPÍTULO 7:

PRUEBAS REALIZADAS

1. PRUEBAS REALIZADAS

1.1. Detección del suelo

En la ilustración 10 muestra el resultado de la primera prueba tras programar la detección del suelo. Se aprecia la nube de puntos capturada, en la cual aparecen puntos en blanco y en rojo.

Los puntos en rojo representan los puntos pertenecientes al plano del suelo. Debido a la disposición de la cámara pueden apreciarse puntos blancos que también pertenecen al plano. Estos puntos no se han detectado correctamente y es necesario ajustar el código implementado.

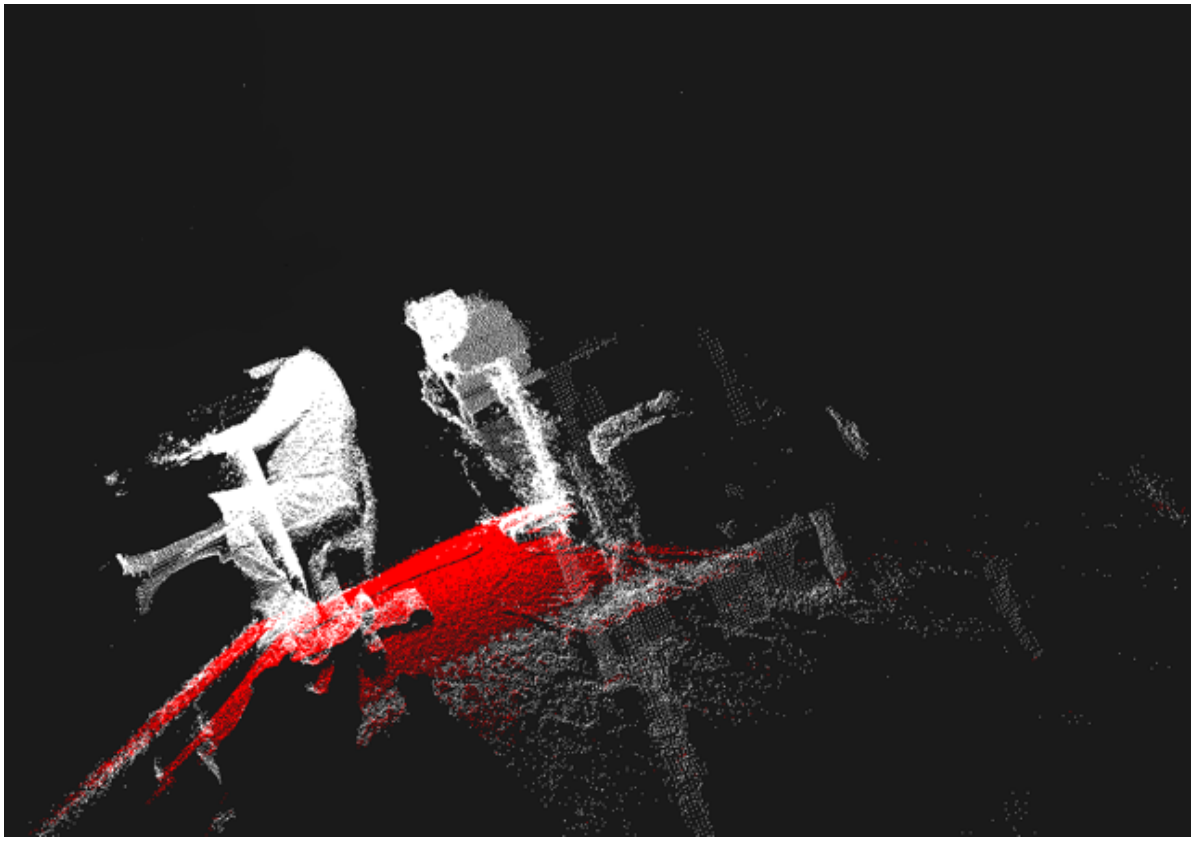


Ilustración 10: Resultado prueba 1 detección de suelo

La siguiente ilustración corresponde a la segunda prueba de detección del suelo. Para esta prueba se colocó el sensor Kinect encima de una mesa.

En esta ilustración 11 el sensor obtiene la imagen de la mayor parte de la estancia. Nuevamente se aprecian puntos en blanco y rojo. Los puntos rojos, nuevamente, corresponden a los puntos del suelo. En este caso se toma como suelo la mesa en la cual se sitúa la Kinect.

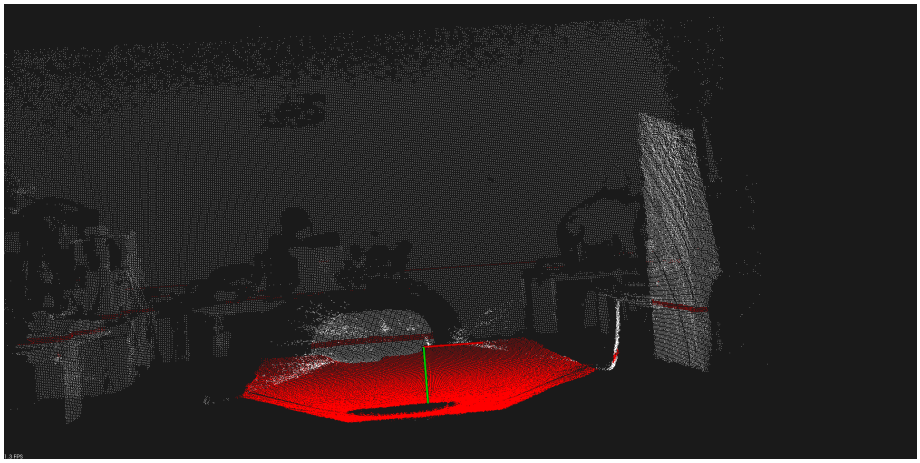


Ilustración 11: Resultado prueba 2 detección de suelo

1.2. Eliminación de los puntos del suelo

Tras completar la detección del suelo, se procede a suprimir los puntos que pertenecen a él. En la siguiente ilustración se puede ver el resultado de la primera prueba de eliminación. Corresponde a la misma imagen que la Ilustración 12. En esta nueva imagen se ve la nube de puntos después de aplicar un filtro 'extract' de la biblioteca PCL. Como se puede apreciar, se han eliminado los puntos pertenecientes al plano del suelo, señalados en rojo en la ilustración 9.

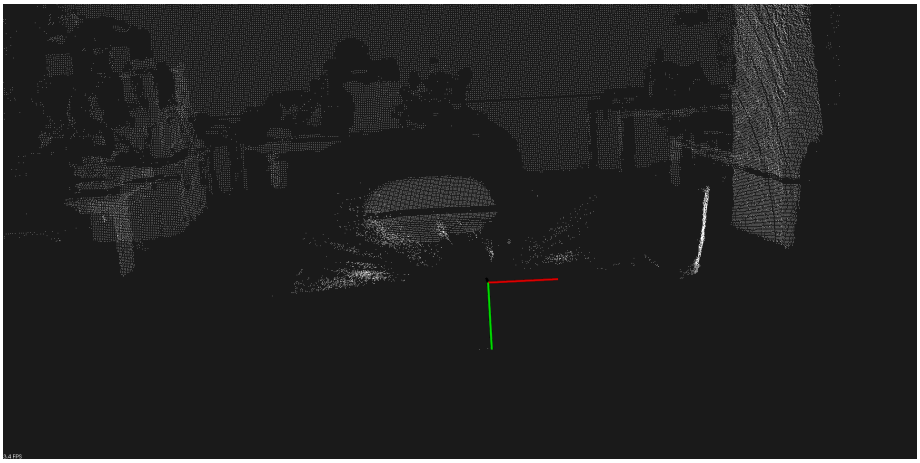


Ilustración 12: Resultado prueba eliminación del suelo

1.3. Prueba completa en el laboratorio

En esta apartado se muestra el resultado obtenido tras realizar la primera prueba una vez concluido el desarrollo del proyecto. Aunque esta prueba se realiza en el laboratorio, en lugar del exterior, se puede tomar por bueno el resultado obtenido.

En primer lugar, en la ilustración 13 se muestra la posición de la cámara en el entorno. Se coloca sobre una mesa a 78cm del suelo (distancia medida hasta el objetivo de la cámara).

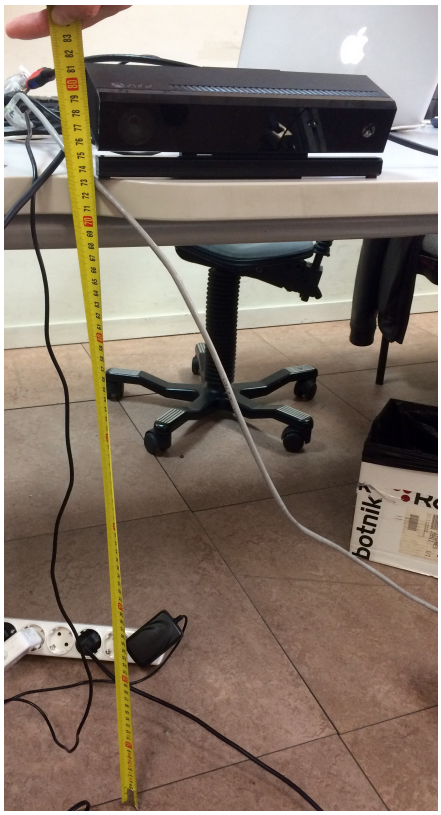


Ilustración 13: Disposición de Kinect para prueba en laboratorio

En esta posición se obtiene una imagen completa del laboratorio que se puede ver en la ilustración 14, donde se muestra la imagen de profundidad obtenida.

Ilustración 14: Imagen de profundidad durante la prueba

En esta ilustración se ve una mesa de escritorio y una silla ubicadas frente a la cámara. Se trata de una imagen de profundidad en la cual las zonas más grises son aquellas que se encuentran más cerca de la cámara, mientas que las zonas más claras están ubicadas a mayor distancia.

Esta imagen de profundidad corresponde con la nube de puntos que se muestra en la ilustración 15. Esta nube será la que se utilice para realizar el escaneo laser.

Ilustración 15: Nube de puntos obtenida durante la prueba

Por último, se muestra en la ilustración 16 el escaneo laser resultante de la imagen de profundidad y nube de puntos anteriores. En esta ilustración se ve como se representan los puntos más cercanos a la cámara.

Los obstáculos más importantes son los más cercanos. Es decir, si en la misma línea de detección se encuentra un punto a 2m de la cámara y otro a 6m, se representará el punto más cercano, es decir, el situado a 2m. El escaneo laser se realiza de esta manera porque, en el caso de usar el mapa generado para navegación, el primer obstáculo a evitar es el más cercano a la cámara y por tanto el más cercano al coche.

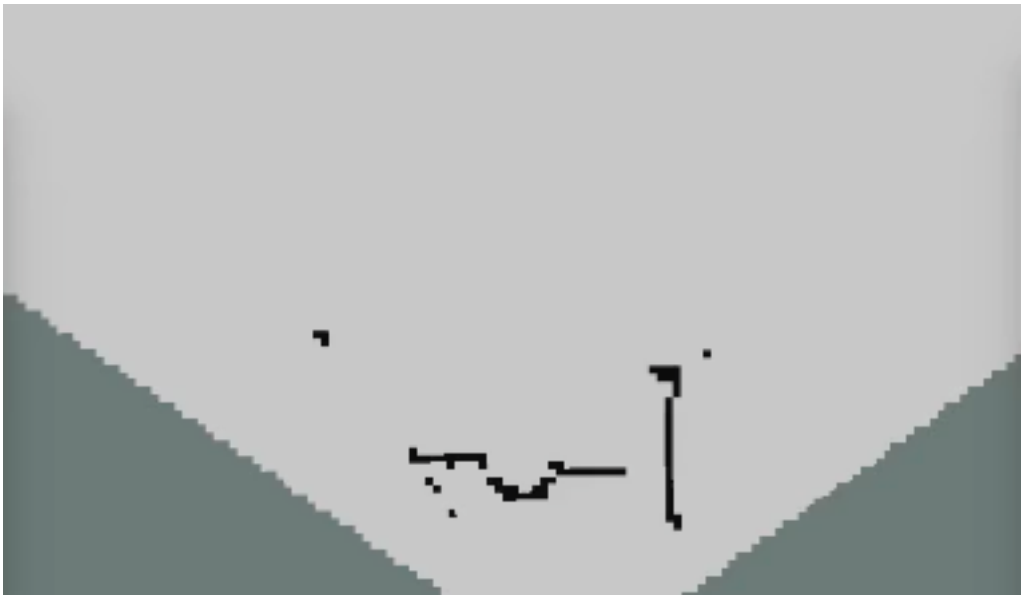


Ilustración 16: Mapa obtenido durante la prueba

1.4. Prueba completa en el laboratorio II

La segunda prueba, realizada en el laboratorio, tuvo como fin comprobar si el mapa resultante cambiaba si aparecía un obstáculo de forma repentina. El obstáculo elegido es una persona que se introduce en el campo de detección de la cámara.

En la ilustración 17 se puede ver cómo fue variando el mapa a medida que la persona pasaba delante de la cámara.

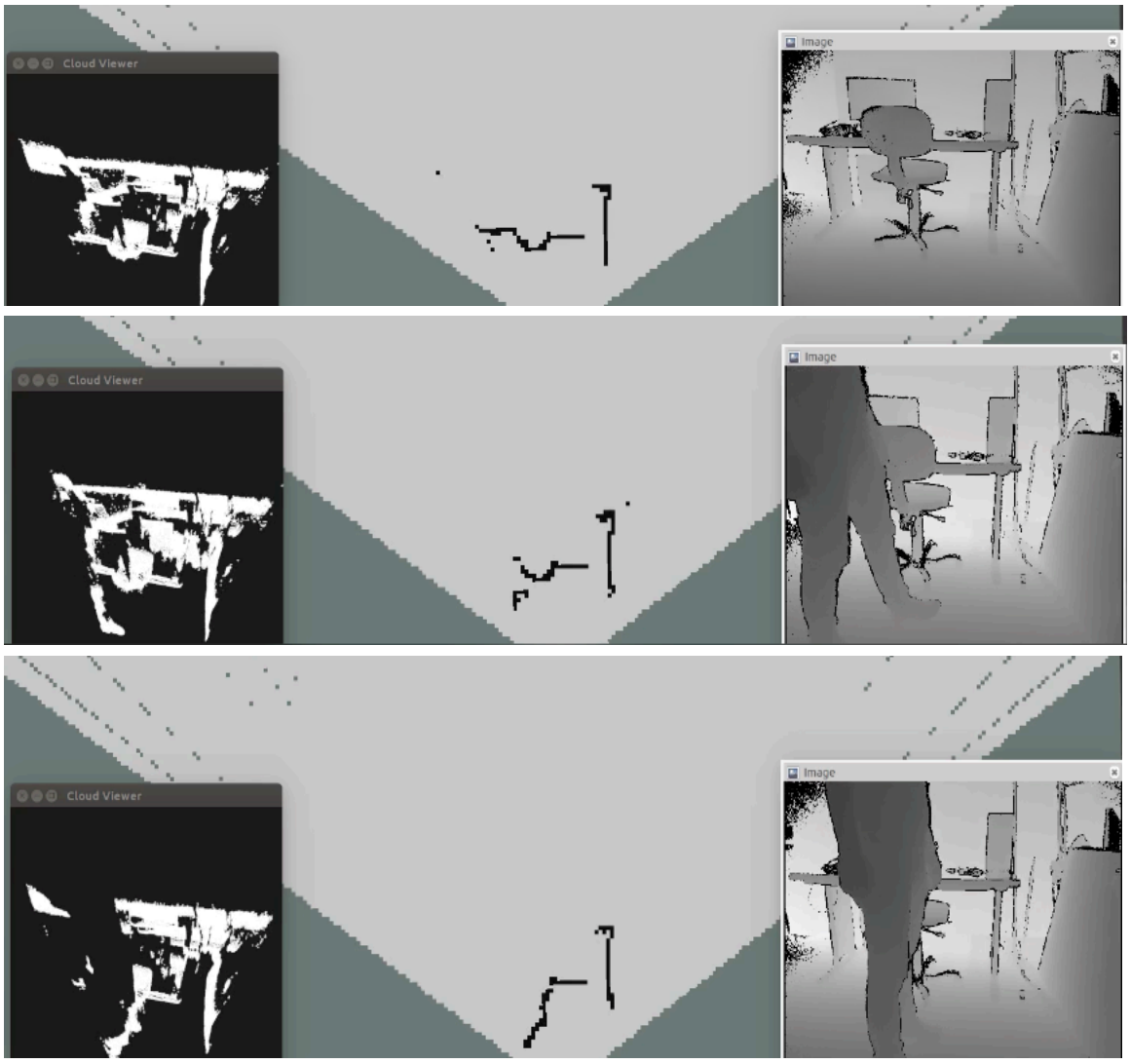


Ilustración 17: Prueba de detección de personas

En la primera imagen se vuelve a ver representada la silla y el escritorio. A medida que la persona aparece en el mapa, el resultado va cambiando. Se representan los puntos pertenecientes al obstáculo situado entre la mesa y la cámara. Por ello en la tercera imagen solo se ve el pico derecho de la mesa, además de la pared. Esto es porque los puntos de la nube más cercanos son los que definen la persona que se ve en la imagen de profundidad.

1.5. Prueba en exteriores

Con el fin de probar el programa en un entorno real se graba una secuencia con Kinect en exteriores. Se generan distintos archivos .bag que serán ejecutados en ROS y se obtendrá la nube de puntos necesaria.

Para poder realizar esta prueba se necesita tener en ejecución a través de ROS:

- Nodo Kinect_cloud que se ha desarrollado para generar el escaneo laser a partir de una nube de puntos dada por la cámara. El programa se ejecuta usando la siguiente línea de comandos:

```
rosrun kinect_cloud kinect_cloud
```

- La secuencia grabada en exteriores. Se ejecuta a través de la herramienta rosbag desde un archivo .bag. Para ejecutar la secuencia se debe acceder desde la terminal de Ubuntu al directorio en el cual se han guardado los archivos .bag. Una vez en este directorio se ejecuta la siguiente sentencia:

```
rosbag play [nombre_archivo].bag
```

- El programa RVIZ que permite visualizar tanto la nube que se obtiene como el escaneo laser generado. Para iniciar este programa se ejecuta la siguiente sentencia en la terminal:

```
rosrun rviz r viz
```

1.5.1. Configuración RVIZ para visualización

En RVIZ se debe configurar tanto la visualización de la nube de puntos como la del escaneo laser – laserscan.

Para poder visualizar la nube de puntos se debe agregar un display de tipo 'PointCloud2'. Se selecciona como nube de puntos, en el apartado topic, la nube publicada por el nodo 'kinect_cloud'. El display queda configurado tal y como se muestra en la ilustración 16.

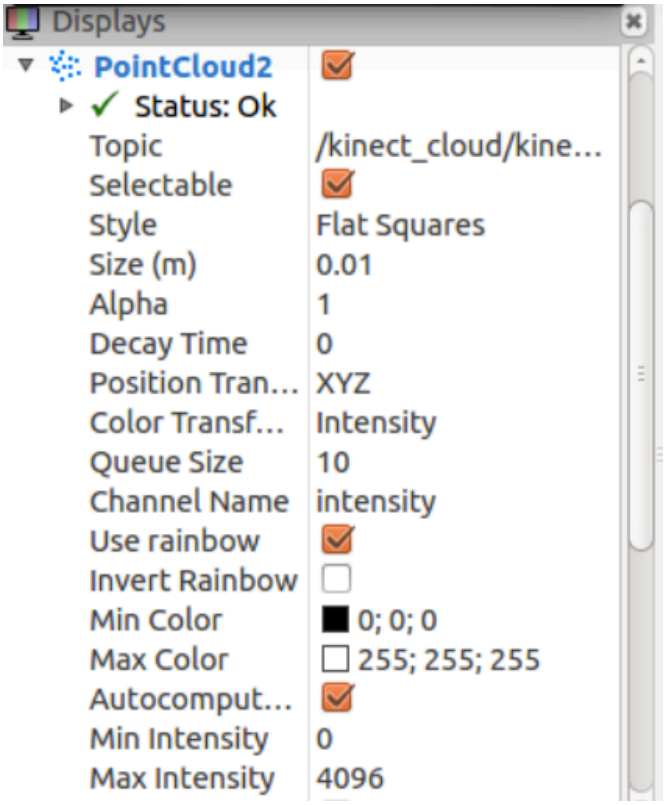


Ilustración 18: Configuración Display de tipo PointCloud

El siguiente paso es configurar la forma de visualizar el escaneo laser. Para ello se usa un display de tipo 'LaserScan'. Se selecciona como topic el escaneo laser que se publica por medio del nodo 'kinect_cloud'. Una vez hecho esto, el display queda configurado de la siguiente manera:

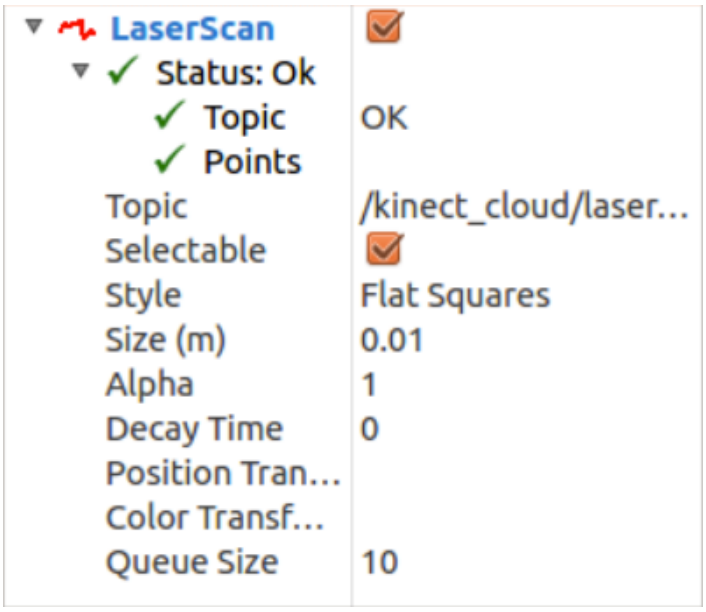


Ilustración 19: Configuración Display de tipo LaserScan

1.5.2. Ejecución y visualización

Las pruebas en exteriores se han realizado en el Campus de Leganés de la Universidad Carlos III de Madrid. Concretamente, se han grabado las secuencias entre los edificios Agustín de Betancourt y Juan Benet. En la siguiente ilustración se puede ver la zona donde se han realizado las pruebas. En verde queda señalado el edificio Juan Benet y en rojo el edificio Agustín de Betancourt.

Ilustración 20: Plano Campus de Leganés

En la ilustración 21 se puede ver en detalle la zona entre los dos edificios, donde se han grabado las secuencias para las pruebas.

Ilustración 21: Zona de pruebas

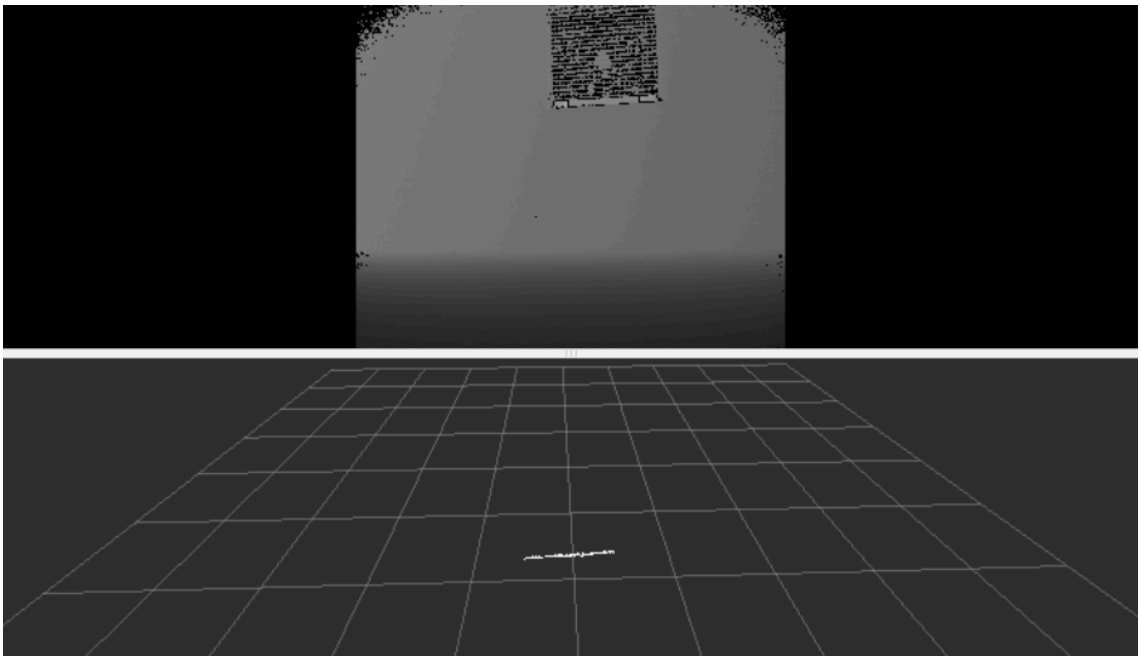


Ilustración 22: Detección de pared

En la ilustración 22, se puede ver el resultado de la detección de una pared de forma frontal. La representación en el mapa es una línea recta que abarca toda la captura que ha realizado la cámara.

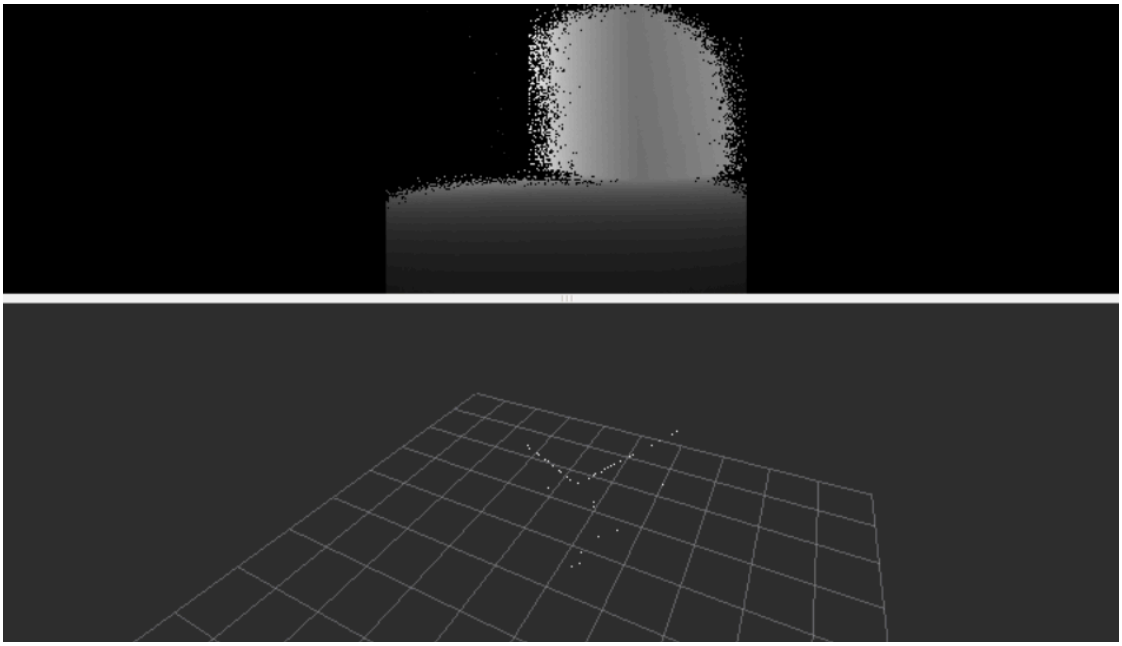


Ilustración 23: Detección de esquina

En la imagen 23 se puede ver como se detecta una esquina. El resultado es la representación de dos líneas perpendiculares. Esta esquina está ubicada a la entrada del edificio 1, concretamente es la esquina exterior de los laboratorios de la zona M de laboratorios.

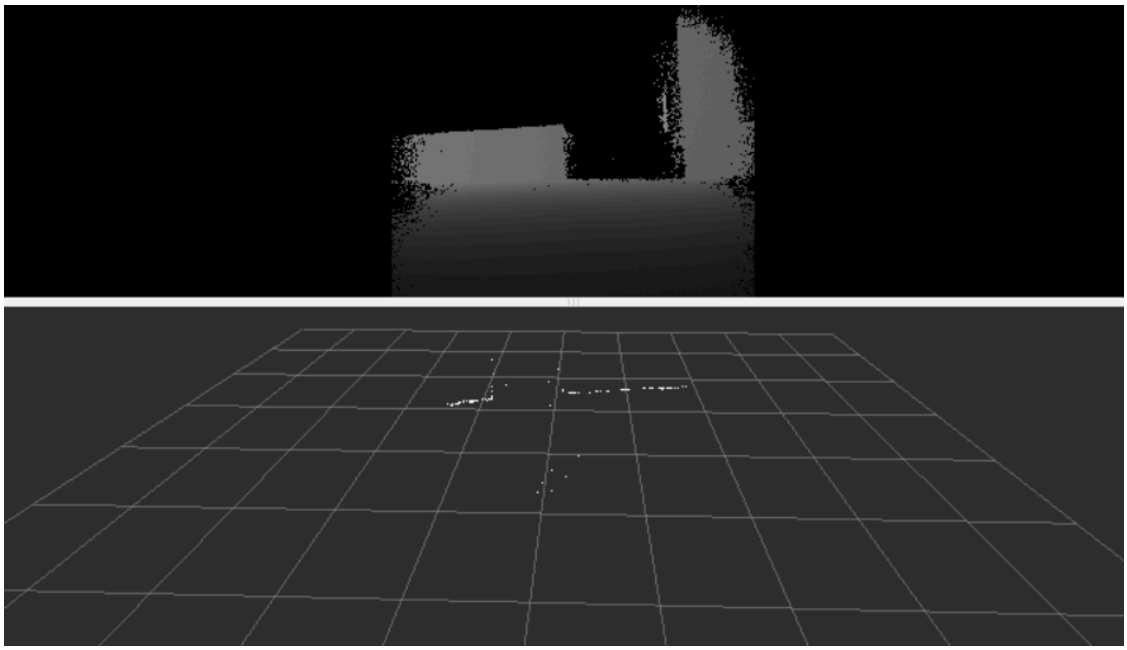


Ilustración 24: Detección de espacio libre

En la ilustración 24 se puede ver como se detecta un espacio libre entre dos pareces. Este espacio se encuentra ubicado entre las dos entradas del edificio Juan Benet.

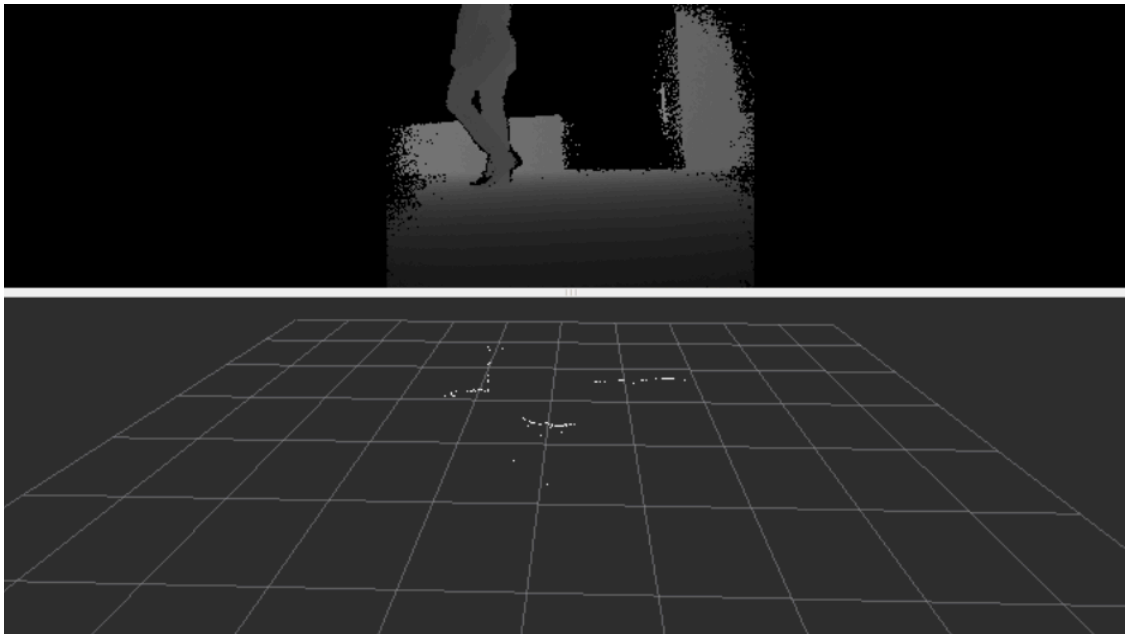


Ilustración 25: Detección de personas I

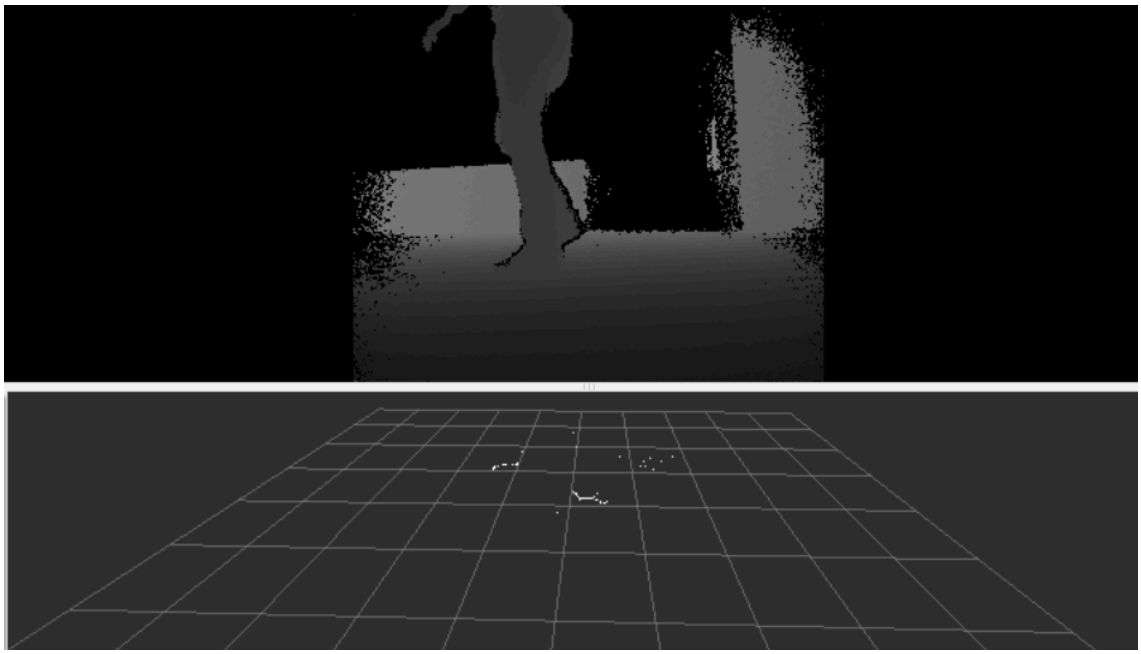


Ilustración 26: Detección de personas II

En las ilustraciones 25 y 26 se aprecia el resultado de la detección de una persona frente a la cámara. Resulta ser una acumulación de puntos que describen el contorno del cuerpo. Se ve como se borra la zona del mapa detrás de la persona,

pues ya no es relevante. Como se explico en puntos anteriores, se representan los obstáculos mas cercanos a la cámara.

CAPÍTULO 8:

CONCLUSIONES Y

PRESUPUESTO

1.1. Conclusiones

Debido al gran crecimiento que ha experimentado el sector del automóvil en relación a sistemas de navegación y coches autónomos, surge la necesidad de desarrollar un algoritmo que genere un mapa bidimensional en el cual se representan los obstáculos encontrados en el entorno.

Es programa implementado da como resultado un mapa de tipo LaserScan en el cual se pueden ver los obstáculos cercanos al vehículo, así como el espacio libre por donde puede navegar el coche autónomo. El sistema desarrollado cumple el objetivo de un sistema fiable, barato y, más importante, en tiempo real.

Para la realización de este proyecto se han usado, principalmente, conocimientos de programación en el lenguaje C++, adquiridos en asignaturas de programación. Además, han sido necesarios conocimientos en navegación y mapeado adquiridos en asignaturas de robótica. Esto último es necesario para crear el mapa resultante.

Además, el proyecto ha requerido aprender nociones básicas de ROS en referencia a ejecutar nodos y envío de mensajes entre ellos. Además, se ha debido realizar un estudio de las librerías PCL para analizar el entorno por medio de nubes de puntos. Ésta es una de las partes más importantes del proyecto.

Como conclusión a este proyecto, es importante destacar los conocimientos en análisis de imagen, nubes de puntos y mapeado en tiempo real. Además de haber aprendido a desarrollar y dirigir un proyecto desde cero.

1.2. Aplicaciones

Tras haber visto en que consiste este proyecto y haber entendido su funcionamiento y las herramientas utilizadas, se pueden analizar las posibles aplicaciones que se le podrán dar.

La aplicación más directa es utilizarlo en coches autónomos como sistema para generar trayectorias y rutas de navegación. Se trata por tanto de un complemento a los sistemas de navegación actuales, que servirá para evitar obstáculos y permitir una conducción segura.

Debido a que los coches autónomos están aún en desarrollo, este proyecto puede tener una aplicación inmediata en el sector del automóvil si se usa como ayuda a la conducción.

Instalando este sistema en un coche convencional, puede usarse como un sistema de aviso al conductor en situaciones meteorológicas adversas. Es decir, conducción nocturna, bajo lluvia intensa o incluyo niebla. Aunque no se ha probado que funcione en estas condiciones, podría ser el próximo objetivo del proyecto.

1.3. Presupuesto

CODIGO	UNIDADES	DESCRIPCIÓN	PRECIO UNITARIO	PRECIO TOTAL
1		**CAPITULO 1. Hardware**		
1.1	1	Dispositivo Kinect 2.0. Incluye cables de comunicación	95,95 €	95,95 €
1.2	1	Placa de desarrollo Odroid XU4	83,14 €	83,14 €
1.3	1	Transformador 12V para Odroid	13,90 €	13,90 €
1.4	1	Memoria eMMC 16GB	30,33 €	30,33 €
1.5	1	Microsoft 97-0004 para PC	39,47 €	39,47 €
		SUBTOTAL CAPITULO 1		**262,79 €**
2		**CAPITULO 2. COSTE PERSONAL**		
2.1	460	Precio por hora del ingeniero proyectista	90,00 €	41.400,00 €
		SUBTOTAL CAPITULO 2		**41.400,00 €**
		TOTAL		**41.662,79 €**

Tabla 2: Presupuesto

ANEXOS

ANEXO I: Entender nodos y topics: ejemplo Turtlesim

Este es un ejemplo con el que se pueden entender los conceptos de nodo, topic y mensajes de ROS.

Para entender correctamente como trabajan los nodos y para que se usan los topics, vamos a realizar el ejemplo de turtlesim, que corresponde a uno de los tutoriales básicos de la wiki de ROS.

Suponemos que ROS esta instalado correctamente – la explicación de cómo se instala se lleva a cabo en la parte de configuración.

Para comenzar a ejecutar ROS, abrimos una terminar y ejecutamos:

```
$ roscore
```

Es necesario que esta terminal este abierta siempre que estemos trabajando con ROS. Seguidamente abrimos dos terminales más para ejecutar los nodos.

En la primera terminal ejecutaremos el nodo turtlesim_node. Al ejecutar este nodo veremos una pantalla con una tortuga. Para ello ejecutamos en la terminal:

```
$ rosrun turtlesim turtlesim_node
```

En la segunda terminal vamos a ejecutar el nodo turtle_teleop_key. Este nodo permite mover la tortuga anterior con las flechas del teclado. Para ejecutarlo:

```
$ rosrun turtlesim turtle_teleop_key
```

Con los dos nodos ejecutándose, si pulsamos una flecha podemos ver que la tortuga se mueve en esa dirección. El nodo teleop_turtle publica un topic con la información del movimiento. El nodo tutlesim está suscrito a este topic y usa los datos contenidos en el para realizar el movimiento de la tortuga.

Se puede ver el funcionamiento gráficamente ejecutando en una terminar:

```
$ rosrun rqt_graph rqt_graph
```

ANEXO II: METODO RANSAC

1. Definición

El método RANSAC es un método iterativo para calcular los parámetros de un modelo matemático de un conjunto de datos observados que contiene valores atípicos. Es un algoritmo no determinista en el sentido de que produce un resultado razonable sólo con una cierta probabilidad, mayor a medida que se permiten más iteraciones. El algoritmo fue publicado por primera vez por Fischler y Bolles SRI International en 1981.

Los datos consisten en "inliers", es decir, los datos cuya distribución se explica por un conjunto de parámetros del modelo, aunque pueden estar sujetos a ruido, y "valores atípicos", que son datos que no encajan en el modelo. Los valores atípicos pueden provenir, por ejemplo, de valores extremos del ruido o de mediciones erróneas o hipótesis incorrectas sobre la interpretación de los datos. RANSAC también asume que, dada un conjunto de inliers (generalmente pequeño), existe un procedimiento que puede estimar los parámetros de un modelo que explica de manera óptima o se ajusta a esta información.

2. Ventajas y desventajas de RANSAC

Una ventaja de RANSAC es su capacidad para hacer una estimación robusta de los parámetros del modelo, es decir, se pueden estimar los parámetros con un alto grado de precisión, incluso cuando están presentes en el conjunto de datos de un número significativo de valores atípicos. Una desventaja de RANSAC es que no hay tiempo máximo para calcular estos parámetros (excepto agotamiento). Cuando el número de iteraciones calculadas se limita a la solución obtenida puede no ser un resultado óptimo, y puede incluso no ajustarse a los datos. De esta manera RANSAC ofrece una solución de compromiso; mediante el cálculo de un mayor número de iteraciones se incrementa la probabilidad de encontrar un modelo razonable.

Por otra parte, RANSAC no siempre es capaz de encontrar la configuración óptima incluso para conjuntos moderadamente contaminados y por lo general tiene un rendimiento pobre cuando el número de inliers es inferior al 50%.

Optimal RANSAC se propuso para manejar estos dos problemas, es por eso que es capaz de encontrar el conjunto óptimo para conjuntos muy contaminados, incluso para una relación de inlier debajo del 5%. Otra desventaja de RANSAC es que requiere el establecimiento de umbrales de problemas específicos.

BIBLIOGRAFÍA

"ODROID", Odroid Downloads, 2016. [Online]. Enlace: http://com.odroid.com/sigong/nf_file_board/nfile_board.php. [Acceso: Feb- 2016].

W. Woodall, "jade/Installation/Ubuntu - ROS Wiki", Wiki.ros.org, 2016. [Online]. Enlace: http://wiki.ros.org/jade/Installation/Ubuntu. [Acceso: Feb- 2016].

"ROS/Tutorials - ROS Wiki", Wiki.ros.org, 2016. [Online]. Enlace: http://wiki.ros.org/ROS/Tutorials. [Acceso: Mar- 2016].

"Downloads - Point Cloud Library (PCL)", Pointclouds.org, 2016. [Online]. Enlace: http://www.pointclouds.org/downloads/linux.html. [Acceso: Mar- 2016].

"Point Cloud Library (PCL): pcl::SACSegmentation< PointT > Class Template Reference", Docs.pointclouds.org, 2016. [Online]. Enlace: http://docs.pointclouds.org/1.7.0/classpcl_1_1_s_a_c_segmentation.html. [Acceso: May- 2016].

"Point Cloud Library (PCL): Module filters", Docs.pointclouds.org, 2016. [Online]. Enlace: http://docs.pointclouds.org/trunk/group__filters.html. [Acceso: Apr- 2016].

T. Wiedemeyer, "code-iai/iai_kinect2", GitHub, 2016. [Online]. Enlace: https://github.com/code-iai/iai_kinect2. [Acceso: Mar- 2016].

P. Bovbel, "pointcloud_to_laserscan - ROS Wiki", Wiki.ros.org, 2015. [Online]. Enlace: http://wiki.ros.org/pointcloud_to_laserscan. [Acceso: Jun- 2016].

P. Marín-Plaza, J. Beltrán, A. Hussein, B. Musleh, D. Martín, A. Escalera, J. Armingol, "Stereo Vision-based Local Occupancy Grid Map for Autonomous

Navigation in ROS", Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2016) - Volume 3: VISAPP, pages 703-708.